Lara Raquel Saraiva Luís

Licenciada em Engenharia Informática

Repairs of Databases with Null Values

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador : João Leite, Professor Auxiliar,
Universidade Nova de Lisboa

Co-orientador : Martin Slota, Investigador,
Universidade Nova de Lisboa

Júri:

Presidente: Prof.ª Carla Maria Gonçalves Ferreira,
Universidade Nova de Lisboa

Arguente: Prof. Vítor Manuel Beires Pinto Nogueira,
Universidade de Évora

Vogal: Prof. João Alexandre Carvalho Pinheiro Leite,
Universidade Nova de Lisboa

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Novembro, 2013

Repairs of Databases with Null Values

Acknowledgements

First of all, I want to express how thankful I am to belong to this institution, Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa, that since the first moment, made me feel at home. It helped me to easily face a new stage of my life, making me grow professionally and personally, and allowed me to have access to the best education and best teachers that I could get. Thank you to all the people that participated in this process during these years.

With respect to this dissertation specifically, I want to express my appreciation to my advisors, João Leite and Martin Slota, that guide me through this process. To João Leite for being one of the best teachers that I had and making me gain interest in databases in the beginning of the course, and to Martin Slota for the patience in the long online meetings. Thank you for the advices, for the requirements, for the knowledge and for allowing me to develop a subject that I always wanted to work on.

This work was partially supported by FCT project ASPEN - Answer Set Programming with BoolEaN Satisfiability (PTDC/EIA-CCO/110921/2009).

I also want to thank to the colleagues for making me feel integrated, part of the family, for their support, sense of humor, for hearing my complaints. They allowed me to not walk through this path alone. Thanks for the discussions, for the jokes, for the knowledge sharing, for being the best fellows I could get. Main thanks to Laura Oliveira, Andy Gonçalves, Joana Roque, Helder Martins and João Silva. Also, thanks to Bruno Filipe Faustino for having patience for reading the document, for the support and for the long conversations.

I cannot forget to mention my since ever best friend for being there everyday and give me motivation to keep going, to achieve my goals, and that always believed in me, Íris Grilo.

At last but not least, I want to thank to my parents for being there, for always supporting me and ask me how the things were going. Thanks for the real concern.

Abstract

Databases store information that is intended to model the real world and to help in modeling, they use constraints that shape the information according to the world view. However, when a new constraint is defined, the data contained in the database may not respect it and so the database should be repaired. Those repairs are made by adding, removing or updating tuples, making as few modifications as possible to satisfy the constraints. In order to determine the repairs of a database with respect to new constraints, there are already some available approaches that provide a solution. But databases also need to contain information that is absence, which is represented through null values. Null values are not regular values and they represent information that is missing or unknown. When using null values, there is no consensus in the literature on how to interpret them when checking constraint satisfaction. Also, there is not a practical implementation to do the repairing regarding null values.

In this document, we study the problem of dealing with null values in the repairing process and propose a (both practical and theoretically sound) solution for this problem including the definition of semantics for null values to achieve constraint satisfaction, and how to proceed to make the databases repairs, ending with a practical implementation of the proposed solution using Answer-set Programming.

Keywords: databases, inconsistency, constraints, repairs, minimal change, null values, answer-set programming.

Resumo

As bases de dados guardam informação que pretende modelar o mundo real e para ajudar na modelação, estas usam restrições que modelam a informação de acordo com a visão do mundo. No entanto, quando uma nova restrição é definida, os dados contidos na base de dados podem não respeitar essa restrição e então é necessário fazer a reparação da base de dados. Essas reparações são conseguidas adicionando, removendo ou actualizando tuplos, realizando o mínimo de alterações possíveis até que as restrições sejam cumpridas. Para determinar as reparações de uma base de dados relativamente a novas restrições, existem já algumas abordagens que fornecem uma solução. No entanto, as bases de dados também precisam de conter informação que está ausente, sendo representada por valores nulos. Os valores nulos não são valores normais e representam informação que está em falta ou é desconhecida. Quando se usam valores nulos, não há um consenso na literatura na forma de interpretar essa informação para a satisfação das restrições. Além disso, não existem implementações que realizem reparações que considerem valores nulos.

Neste documento, estudamos o problema de lidar com valores nulos na reparação de base de dados e propomos uma solução (prática e teórica) para este problema, incluindo as semânticas possíveis na presença de valores nulos para a satisfação de restrições, e como proceder para realizar as reparações da base de dados, culminando na implementação prática da solução proposta fazendo uso de Answer-set Programming.

Palavras-chave: bases de dados, inconsistência, restrições, reparações, minimalidade, valores nulos, answer-set programming.

Contents

List of Figures

List of Tables

1

Introduction

From the earliest days of computers, storing and manipulating data has been a major application focus [RG02]. Today, we have a lot of organizations that keep their information available due to computers. For example, libraries keep information about available books; hospitals keep information about patients and appointments; stores keep information about products and suppliers. In order to maintain that information and keep it organized, we use databases. With databases, we group data according to its purpose and characteristics. For example, it might make sense to group together the products of a specific store because all of them have a name, a price and a category.

In order to store and retrieve database information we use a Database-Management System (DBMS). A DBMS provides methods to maintain data safety, namely to give some guarantees about integrity of stored data so that we can trust in it as much as possible, provides concurrent access to data, crash recovery from system failures and efficient data access [RG02]. Before DBMSs, file-processing systems presented many difficulties which prompted the development of database systems [SKS10].

Underlying the structure of a database are the data models that describe the design of the database at different levels [SKS10]. In 1971, Edgar Codd showed how a relational model could be used to define and structure large databases of organizations, being best fitted to their needs [Eri09]. The relational model uses a collection of tables to represent information. A table is composed of a set of columns and a set of records. Each record (an entity) has a fixed number of attributes (the properties of the entity) that are the columns of the table. For example, if we need to store information related to a collection of products, then each record represents a different product and each attribute represents a different characteristic of the product, e.g. the name and the category.

The relational model was an important step towards what we have today. It provided

a more logical way of storing information by making use of relationships between tables that, together with its conceptual simplicity, has led to its widespread adoption. Today the majority of database products are based on this model [SKS10].

However, the relational model just by itself cannot model the relationships between tables using the real world functioning of an organization. Databases have to guarantee, as much as possible, data consistency in order to contain reliable information and be useful to the organization that needs it.

1.1 Integrity constraints

For the purpose of having an accurate representation of the context that is being modeled, we need to impose some restrictions according to the modeled context on data so that it does not represent unrealistic situations. For example, returning to the products example, we would only allow to have products whose price is a value that is greater then zero, since we cannot have negative prices. Also, if we consider the books of a library, we want each book to have an author from our own list of authors. To achieve that, we use integrity constraints (ICs).

Constraints are formulated by the database designer to express the conditions that data must follow in order to be consistent with the context that is being modeled. Constraints are used to validate the changes to ensure that the database remains consistent with the world view. So, data is meaningful only when it reflects those constraints.

DBMSs are responsible for ensuring that the ICs are not violated. Consequently, to avoid their violation, currently existing DBMSs reject updates that make the database inconsistent regarding the ICs, which may not be expected by the user.

Databases may become inconsistent due to some factors like the integration of different types of databases or even if the user specifies new ICs. In the last case, data may not be according with the new IC and so, the user may want to know that. It would be more interesting to the user to know what she can do to solve the problem. For example, instead of just rejecting the IC, the user may want to just do some small modifications on data such as insertions, deletions of updates in order to make it consistent with the ICs and so keep the data and keep the new IC.

1.2 Null values

There are situations where we need to represent missing or unknown information because we do not know it. For example, returning to the product example, we may have all the information about a product but we do not know in which category it fits. Attributes that represent unknown information are represented by null values. They should be considered when we talk about databases since we often want to represent some part of the information that we do not know.

This type of values differ from the other ones since they do not represent a specific known information, but the absence of it, and they can have different meanings since we do not know if the information is really unknown or if the value does not exist. Also, we cannot treat these values like the other ones: we cannot compare them in the same way, for example. Regarding the products example, we may consider that two products have unknown suppliers that may represent the same supplier. Also, we can consider that, as they are unknown, they represent different suppliers and so when we are comparing two unknown values, we consider that the result of the comparison is false. This is why they can be difficult to handle so, we need to assign meaning to them in all situations in order to deal with their interpretations ambiguity.

1.3 Problem

When we add a new IC how can we make the database consistent when the existing data does not respects it? Most DBMSs just reject the whole update that turns the database inconsistent that is, that does not satisfy the constraints. However, we could drop all the database so that when we add new data it would respect the new IC but this would make us lose all data every time we want a new IC, which is obviously unsatisfactory. Instead, we can remove some tuples that are violating the IC or maybe insert other ones instead of just rejecting the updates or delete all data. It would be better to do as few operations as possible that would make the database consistent, reducing the impact on the database. This set of operations provokes changes on data and the set of these changes is called repair. A repair originates a database instance that is consistent with the ICs and that instance minimality differs from the original database.

Repairs can be computed in two different circumstances. They can be computed when a new IC is added, then one of the repairs is chosen and it replaces the database (Database Repairing here represented as DR). Or, they can be computed when the user makes a query to the database and so repairs are used to answer the query (Consistent Query Answer here represented as CQA), and no change occurs in the database.

For the same database, there can be more than one alternative repair as illustrated in the following example, based on [BB04].

Example 1 (Simple employees example)**.** Consider that we have a table that stores information about employees and that we want to impose a constraint that states that the same employee cannot have more than one date of birth. We distinguish the employees by their name.

So, using First-Order Logic and considering the $Employee$ table, the IC can be expressed as:

$$\forall_{Name,DateOfBirth,Salary,Position,DateOfBirth2,Salary2,Position2}$$

$$(Employee(Name, DateOfBirth, Salary, Position) \wedge$$

$$Employee(Name, DateOfBirth2, Salary2, Position2) \rightarrow DateOfBirth = DateOfBirth2)$$

Which states that from now on each employee can only have a single date of birth.

Suppose that the contents of the table *Employee* are as follow:

Name	*DateOfBirth*	*Salary*	*Position*
John	24-03-1990	*55.000*	*manager*
Peter	27-05-1985	*50.000*	*manager*
John	24-03-1991	*60.000*	*manager*

Table 1.1: Employees table

As we can see, the IC is not respected since there are two equal values for *Name* attribute and the corresponding values for *DateOfBirth* are different. We could delete all the data and the database would be consistent with respect to the IC or we can change it as little as possible in order to satisfy the constraint.

Two alternative (minimal) repairs are:

Name	*DateOfBirth*	*Salary*	*Position*
John	24-03-1990	*55.000*	*manager*
Peter	27-05-1985	*50.000*	*manager*

Table 1.2: Repair 1 of employees table

Name	*DateOfBirth*	*Salary*	*Position*
Peter	27-05-1985	*50.000*	*manager*
John	24-03-1991	*60.000*	*manager*

Table 1.3: Repair 2 of employees table

By choosing one of the repairs, we get a database that satisfy the ICs. However, none of them is better than the other, the user can just choose the one that she prefers.

Regarding the previous example, with CQA and the query

```
select * from employee.
```

we get the following content of table *Employee* :

Name	*DateOfBirth*	*Salary*	*Position*
Peter	27-05-1985	*50.000*	*manager*

Table 1.4: Employees table - CQA

The resulting table just includes the data that is present in all the repairs that is, that is always consistent. With DR, the content of the table corresponds to one of the presented repairs, differing from CQA. However, if the IC was like:

$$\forall_{Name,DateOfBirth,Salary,Position}(Employee(Name, DateOfBirth, Salary, Position) \wedge$$

$$Salary > 50.000)$$

Considering CQA and the query

```
select * from employee.
```

we would get:

Name	*DateOfBirth*	*Salary*	*Position*
John	24-03-1990	*55.000*	*manager*
John	24-03-1991	*60.000*	*manager*

Table 1.5: Employees table - CQA

With respect to DR, the only (minimal) repair possible is:

Name	*DateOfBirth*	*Salary*	*Position*
John	24-03-1990	*55.000*	*manager*
John	24-03-1991	*60.000*	*manager*

Table 1.6: Employees table - DR

That is, both have the same results.

In order to solve the problem of adding new ICs that are not respected by the database content, some work has been developed to determine the repairs with respect to the new ICs. Most approaches to the problem are only theoretical, offering a formalization using First-Order Logic (FOL) and some of them also offer a transformation of the formalization into a logic program. There is also some practical work done as in [Alv11] where an application was developed that allows to define ICs and computes repairs using the Answer-Set Programming (ASP).

However, we cannot forget that there are some special values, the null values. A null value can have multiple interpretations. However, in commercial DBMSs, all the interpretations are represented with a single constant and this is also our approach.

Null values can be difficult to handle when it comes to the satisfaction of ICs, since we cannot manipulate them in the same way as the other attributes, as we can see in the following example:

Example 2 (Simple employees example with null values)**.** Considering again the table of Example 1 but now with null values:

Name	*DateOfBirth*	*Salary*	*Position*
null	24-03-1990	*55.000*	*manager*
Peter	27-05-1985	*50.000*	*manager*
null	24-03-1991	*60.000*	*manager*

Table 1.7: Employees table

In this case, determining whether the table satisfies the constraint depends on how we interpret the null values within it. If we treat a null value the same way as any other

value, we can say that the table is inconsistent since there are two null values from $Name$ attribute that correspond to different $DateOfBirth$ values. However, they are not regular values. We could have the indication that the range of available values for the $Name$ attribute are only the values that are already in that database's tuples and, in that case, we surely have an inconsistency, because by having one null value we know that it represents one of the values that is already there. Without indicating the range the value that is missing or not known may be one of the values that are already in the table. On the other hand, since a null value represents unknown information, it may represent two different names because we do not know which names are and so we could not say that the IC is being violated, they may just be two different names that are unknown.

That is one of the problems of having null values when repairing a database. It is important to adapt the strategies of satisfaction of ICs and, consequently, the repairing process to null values. We illustrate this in the following example extracted from [CB07].

Example 3 (Using null values when inserting tuples)**.** Consider that we have a database D that contains two tables, P and R. We want to impose a constraint that states that for every value x of P, there is also the same value for the first attribute of R. So, using First-Order Logic and considering the database D where $D = \{P(a, null), P(b, c), R(a, b)\}$, the IC can be expressed as:

$$\forall_{xy}(P(x, y) \rightarrow \exists_z R(x, z))$$

A possible repair is $D_1 = \{P(a, null), R(a, b)\}$, obtained by deleting $P(b, c)$. It removes the tuple that is problematic, solving the inconsistency problem. The other alternative repair is $D_2 = \{P(a, null), P(b, c), R(a, b), R(b, null)\}$. D_2 is obtained by inserting a new tuple $R(b, null)$. It inserts another tuple so that the first tuple is not problematic anymore, matching with the IC. For instance, when inserting new tuples to restore consistency, it is possible to choose null values to represent the attributes we do not know and that admit multiple values that could satisfy the given IC. So, we can use a null value instead of choosing one of multiple attribute values. If we do not choose a null value, we generate an alternative repair for each value of the domain of the attribute, generating a lot of alternative repairs. In this case, the null value could be replaced by any value of the domain like a, b or c.

Due to the difficulties associated with null values, namely, how to represent missing or unknown information, how to interpret a null value in each example, how to compare them, when to accept a null value as consistent with respect to a specific IC, there are not many approaches that consider them when repairing databases. Some of them just consider that these values are present but are ignored and do not specify a particular treatment to deal with them. Also, the only existing implementation ([Alv11]) only deals with databases that do not have null values. There are some approaches in the literature in order to find the most appropriate interpretation when dealing with null values but there is no consensus, there is no standard. Some semantics are introduced in SQL standard [Mel03] that specifies how to verify if each row respects an IC for each of the

presented semantics and there are also approaches that try to complement these semantics, resulting in some alternatives. Two of the approaches that suggest some semantics to deal with null values are [BB04], where it is stated that "since we do not have precise information about them [null values], we will consider that no inconsistencies arise due to their presence" and [BB06], but they do not provide an implementation, where "(...) we propose first a precise semantics for IC satisfaction in a database with null values that is compatible with the way null values are treated in commercial database management systems".

Taking the difficulties of consider null values into consideration, and the fact that there are no approaches that repair databases with null values, we highlight our motivations by analyzing the problems associated with considering null values and we suggest a solution to deal with these values. We define how to represent these values, as we already mentioned, we mention the semantics that are already available and analyze, complement and extend them, in order to determine which is the best semantics according to each IC type by experimenting with the available semantics applied to different examples. These experiments may result in choosing the semantics that apparently fits better to each IC type or to propose another semantics if the previous does not fit as well as expected or is not applicable. Also, we determine how to repair an inconsistent database with null values, introducing the available operations to repair it, the criteria to consider to choose minimal answers. These considerations are approached in a theoretical and practical sense, formally defining the choices made and providing an implementation to allow the user to repair a database according with our solution. To accomplish that, we use FOL to represent the theory in this document, namely the mentioned concepts and the soundness and completeness proofs and then we transform those to ASP in order to represent it in logic programming.

With this, we contribute to the state of the art by providing a more complete analysis of the already existing semantics, by precisely specifying which semantics to use in each different IC and why each one fits better to each IC, that is, how to deal with these values in constraint satisfaction, by providing an application that repairs databases with null values making use of the analyzed approaches and by providing soundness and completeness proofs.

So, the main contributions of this dissertation include:

1. A study of the state of the art including the approaches that do not include null values but that present interesting features in the context of the repairing process, and those approaches that in some way include null values, that is, that do not ignore them, considering some simple semantics or that consider a particular type of ICs when dealing with these values;

2. An approach of repairing databases with null values, considering the different interpretations on multiple examples and situations, different semantics, the allowed operations, the minimal criteria, as opposed to other approaches that treat these

values in very simple way, e.g. the same treatment is given to all circumstances where a null value is used. We highlight the importance of these considerations;

3. An implementation of the suggested solution using logic programming to generate the repairs and making it usable by a common user by providing an interface in Java that allows to specify new ICs and also some other parameters to guide the user through the process.

One of the main results of this dissertation, as mentioned, is an application that allows to, given a database that contains null values and a set of ICs, obtain a new database instance that corresponds to the repair of the original one, taking into account the parameters chosen namely the options related to insertion, deletion and update of tuples, the minimality criteria, the limit number of allowed operations. In the application, we apply the semantics decisions that resulted from the analysis made. For that, we analyze each IC type and extract the main interpretation of each and then apply a semantics according to that. For example, for ICs that compare the tuples of the same table between them in order to see if they are equal, as we consider unknown information, it is difficult argue that the null values of the different tuples represent the same values, then we choose a more flexible semantics. Also, we determine which operations we allow, namely insertions, deletions or updates, or multiple because we may not want to lose a lot of information when repairing a database or to increase a lot the size of it. We need also to specify the criteria to consider when choosing minimal answers in order to cover the main situations, because we may want to minimize the number of operations, for example, or give preference to some operations over the others.

To do the implementation we use ASP because it is very efficient for solving search problems, that in this case is a valuable feature because we need to search for all the possible combinations of operations to make to the database in order to find the one that minimal changes the original database and make it consistent with the ICs. Also, ASP is based on answer set semantics of [GL88] and it is "(...) an attractive paradigm for knowledge representation and reasoning. On the one hand, its popularity is due to the availability of efficient off-the-shelf solvers." [GT09] The solvers, called answer-set solvers, are engines that are aimed to compute the answer sets of these programs. After getting the combinations of operations, the solver then prunes the answers that have tuples that violate at least one IC. Each of these answers sets corresponds to an alternative repair to the database. Knowing this, as we have to determine all the combinations of operations and consider that we can deal with big databases, we may have the problem of getting computations that take a lot of time.

Finally, in order to verify if what we proposed and developed is according to the results that we expect to get, that is, that the answer-sets that result from the ASP program (transformed from FOL) allow to retrieve accurate repairs of the database, we present the soundness and completeness proofs for the theoretical proposal in order to match the formalization with the resulting ASP code. Also, we perform benchmark tests in order to

analyze the influence of some parameters the performance of the application, that may depend of the database to be repaired (as the number of tables considered, the number of tuples, the number of ICs) or the options taken by the user in the application interface (e.g. the limit of operations).

Lastly, we describe how we structure this document by chapters:

Chapter 2 In this chapter, we define the concepts that should be kept in mind in the following chapters. We present main concepts with respect to FOL that is used to present the theoretical part of the dissertation, the concepts that compose an ASP program and database concepts, namely the basics about database structure and ICs;

Chapter 3 In this chapter, we present the state of the art, namely the approaches that have interesting features but that do not consider null values;

Chapter 4 In this chapter we focus on the null values subject, namely, it begins by introducing the approaches that include null values in the repairing process and then it follows for the concerns of considering null values, namely, how to deal with null values when checking consistency and how to treat them in all circumstances;

Chapter 5 In this chapter, we propose a transformation of our theoretical approach to an ASP representation in order to have a program that retrieves the repairs associated to the database and the set of ICs;

Chapter 6 In this chapter, we describe the developed application that makes use of the logic program and that is usable by a common user, and it is where we present the architecture of the application, the available features and the decisions;

Chapter 7 In this chapter, we present the analysis of parameters in order to understand if they influence the performance of the application, that is, the benchmark tests.

2

Preliminaries

In this chapter we formalize all the concepts that are mentioned through the document, namely with respect to FOL, then ASP and at last the database concepts.

FOL and ASP sections are divided in syntax and semantics, separating the way we can express the concepts from their associated meaning.

Databases section is divided into basic concepts, constraints concepts and repair concepts.

2.1 First-order logic

We can separate two main aspects of FOL: syntax and semantics. The syntax specifies the expressions used in FOL. The semantics assigns meaning to those expressions.

The following definitions are based on [Fit96] and partially also on [BL04].

2.1.1 Syntax

We start by defining the alphabet of a first-order language. It is composed of logical and non-logical symbols, which are the types of symbols available to express the intended meaning.

Logical symbols can be divided in:

- Quantifiers: $\forall$ (the universal quantifier) and $\exists$ (the existential quantifier);
- Punctuation: ')', '(' and ',';
- Connectives: $\vee$, $\wedge$, $\neg$;
- Variables: ν_1, ν_2, ...

Non-logical symbols have an associated arity, that is, a natural number indicating how many "arguments" each one takes. They express relationships among a number of constants [Eri09] that correspond to the number of "arguments". These symbols can be divided in:

- Function symbols: set of symbols written in uncapitalized mixed case, e.g.: bestFriend;
  - Constant symbols: function symbols of arity 0.
- Predicate symbols: set of symbols written in capitalized mixed case, e.g.: OlderThan.
  - Propositional symbols: predicate symbols of arity 0.

Logical symbols have a fixed meaning and use in the language. That is, FOL is determined by specifying the non-logical ones, that have an application-dependent meaning and use.

Definition 1 (First-Order Language)**.** FOL is determined by specifying:

1. A finite or countable set **R** of predicate symbols, each of which has a non-negative integer associated with it. If $P \in \mathbf{R}$ has the integer n associated with it, we say P is a predicate symbol with arity n;
2. A finite or countable set **F** of function symbols, each of which has a positive integer associated with it. If $f \in \mathbf{F}$ has the integer n associated with it, we say f is a function symbol with arity n;
3. A finite or countable set **C** of constant symbols.

Here, the first-order language determined by **R**, **F** and **C** is denoted by $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$.

Next, we define the grammar of this language. This includes the definition of term, atomic formulas and formulas.

Definition 2 (Term)**.** The family of *terms* of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ is the smallest set meeting the conditions:

1. Any variable is a *term* of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$;
2. Any constant symbol (member of **C**) is a *term* of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$;
3. If f is a function symbol with arity n (member of **F**) and $t_1, \ldots, t_n$ are *terms* of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$, then $f(t_1, \ldots, t_n)$ is a *term* of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$.

A *term* is *ground* if it contains no variables.

Definition 3 (Atomic Formula)**.** An *atomic formula* of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ is any string of the form $R(t_1, \ldots, t_n)$ where R is a predicate symbol with arity n (member of **R**) and $t_1, \ldots, t_n$ are terms of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$; also $\top$ and $\bot$ are taken to be *atomic formulas* of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$.

For the following definitions, we consider that a binary connective is a connective with arity two which considering the previous presented connectives can be $\vee$ or $\wedge$.

Definition 4 (Formula)**.** The family of *formulas* of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ is the smallest set meeting the following conditions:

1. Any atomic formula of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ is a *formula* of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$;
2. If A is a *formula* of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$, so is $\neg A$;
3. For a binary connective $\circ$, if A and B are *formulas* of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$, so is $(A \circ B)$;
4. If A is a *formula* of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ and ν_1 is a variable, then $(\forall \nu_1)A$ and $(\exists \nu_1)A$ are *formulas* of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$.

It is useful to introduce some abbreviations and conventions. We can add or omit parentheses and use square or curly brackets to improve readability. In case of using predicates or function symbols of arity 0, parentheses can be omitted. Also, we can write $(\alpha \supset \beta)$ instead of $(\neg\alpha \vee \beta)$ and we can write $(\alpha \equiv \beta)$ instead of $(\alpha \supset \beta) \wedge (\beta \supset \alpha)$.

Now we distinguish the variables that are in the scope of some quantifier from those that are not.

Definition 5 (Free and Bound Variables)**.** The *free-variable occurrences* in a formula are defined as follows:

1. The free-variable occurrences in an atomic formula are all the variable occurrences in that formula;
2. The free-variable occurrences in $\neg A$ are the free variable occurrences in A;
3. The free-variable occurrences in $(A \circ B)$, with $\circ \in \{\vee, \wedge\}$, are the free-variable occurrences in A together with the free-variable occurrences in B;
4. The free-variable occurrences in $(\forall \nu_1)A$ and $(\exists \nu_1)A$ are the free-variable occurrences in A, except for occurrences of ν_1.

   A variable occurrence is called *bound* if it is not free.

Definition 6 (Sentence)**.** A *sentence* of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ is a formula of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ with no free-variables occurrences.

2.1.2 Semantics

Here, it is assigned a denotation to all symbols of the language and some concepts as model, entailment, and so on.

To begin, an interpretation determines how to interpret constant, function and predicate symbols with respect to a domain.

Definition 7 (Model)**.** A *model* for the first-order language $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ is a pair $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ where $\mathbf{D}$ is a nonempty set, called the *domain* of $\mathbf{M}$ and $\mathbf{I}$ is a mapping, called an interpretation that associates:

- To every constant symbol $c \in \mathbf{C}$, some member $c^{\mathbf{I}} \in \mathbf{D}$;
- To every function symbol $f \in \mathbf{F}$ with n arity, some n-ary function $f^{\mathbf{I}}$: $\mathbf{D}^n \to \mathbf{D}$;
- To every predicate symbol $P \in \mathbf{R}$ with arity n, some n-ary relation $P^{\mathbf{I}} \subseteq \mathbf{D}^n$.

Since formulas contain variables, we need to assign values to them, using a variable assignment.

Definition 8 (Variable Assignment)**.** An *assignment* in a model $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ is a mapping $\mathbf{A}$ from the set of variables to the set $\mathbf{D}$. We denote the image of the variable ν under an assignment $\mathbf{A}$ by $\nu^{\mathbf{A}}$.

Given an interpretation that gives meaning to the constant and function symbols of the language, and an assignment, which gives values to variables, we have enough information to determine values for arbitrary terms.

Definition 9 (Interpretation of Terms)**.** Let $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ be a model for the language $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$, and let $\mathbf{A}$ be an assignment in this model. To each term t of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$, we associate a value $t^{\mathbf{I},\mathbf{A}}$ in $\mathbf{D}$ as follows:

1. For a constant symbol c, $c^{\mathbf{I},\mathbf{A}} = c^{\mathbf{I}}$;
2. For a variable ν, $\nu^{\mathbf{I},\mathbf{A}} = \nu^{\mathbf{A}}$;
3. For a function symbol f, $[f(t_1, \ldots, t_n)]^{\mathbf{I},\mathbf{A}} = f^{\mathbf{I}}(t_1^{\mathbf{I},\mathbf{A}}, \ldots, t_n^{\mathbf{I},\mathbf{A}})$.

It is also important to specify the concepts of satisfaction of formulas so, before that we define some auxiliary concepts.

Definition 10 (ν-variant)**.** Let ν be a variable. The assignment $\mathbf{B}$ in the model $\mathbf{M}$ is an *ν-variant* of the assignment $\mathbf{A}$, provided $\mathbf{A}$ and $\mathbf{B}$ assign the same values to every variable except possibly ν.

X	Y	¬X	¬Y	X ∨ Y	X ∧ Y
true	**true**	**false**	**false**	**true**	**true**
true	**false**	**false**	**true**	**true**	**false**
false	**true**	**true**	**false**	**true**	**false**
false	**false**	**true**	**true**	**false**	**false**

Table 2.1: Truth table

Definition 11 (Truth Value)**.** Let $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$ be a model for the language $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$, and let $\mathbf{A}$ be an assignment in this model. To each formula Φ of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$, we associate a truth value $\Phi^{\mathbf{I},\mathbf{A}}$ (**true** or **false**) as follows:

1. For the atomic cases,

   $[P(t_1, \ldots, t_n)]^{\mathbf{I,A}} = \mathbf{true} \iff \langle t_1^{\mathbf{I,A}}, \ldots, t_n^{\mathbf{I,A}} \rangle \in P^{\mathbf{I}}$,

   $\top^{\mathbf{I,A}} = \mathbf{true}$,

   $\bot^{\mathbf{I,A}} = \mathbf{false}$,

   $\neg\mathbf{true} = \mathbf{false}$,

   $\neg\mathbf{false} = \mathbf{true}$.

2. $[\neg X]^{\mathbf{I,A}} = \neg[X^{\mathbf{I,A}}]$.

3. $[X \circ Y]^{\mathbf{I,A}} = X^{\mathbf{I,A}} \circ Y^{\mathbf{I,A}}$, with $\circ \in \{\vee, \wedge\}$.

4. $[(\forall\nu)\Phi]^{\mathbf{I,A}} = \mathbf{true} \iff \Phi^{\mathbf{I,B}} = \mathbf{true}$ for every assignment $\mathbf{B}$ in $\mathbf{M}$ that is an *ν-variant* for $\mathbf{A}$.

5. $[(\exists\nu)\Phi]^{\mathbf{I,A}} = \mathbf{true} \iff \Phi^{\mathbf{I,B}} = \mathbf{true}$ for some assignment $\mathbf{B}$ in $\mathbf{M}$ that is an *ν-variant* for $\mathbf{A}$.

Definition 12 (Entailment)**.** A formula Φ of $L(\mathbf{R}, \mathbf{F}, \mathbf{C})$ is true in the interpretation $\mathbf{M} = \langle \mathbf{D}, \mathbf{I} \rangle$, denoted $\mathbf{M} \models \Phi$, if $\Phi^{\mathbf{I,A}} = \mathbf{true}$ for all assignments $\mathbf{A}$.

2.2 Answer set programming

ASP is a form of declarative programming oriented towards difficult, primarily NP-hard, search problems [Lif08]. It solves problems by "reducing them to finding the answer sets" [Gel07]. It is based on the stable model/answer set semantics of logic programming proposed in [GL88], which applies ideas of autoepistemic logic, for example, and in a more general sense, it includes all applications of answer sets to knowledge representation [Bar01].

In ASP, search problems are reduced to computing stable models, which is a responsibility of answer set solvers. These are programs for generating stable models and they use search algorithms [Lif08].

In this section, we can separate two main aspects of ASP: syntax and semantics.

2.2.1 Syntax

The basic building blocks of the language of ASP are function-free first-order terms and atomic formulas (or atoms). We avoid to define atoms because we use the first-order definition of atom. We start by defining the concepts using [JSVdC12] as reference.

Definition 13 (Literal)**.** A *literal* is either an atom t or a negated atom $not\ t$. A *literal* is ground if it does not contain variables.

A logic program (or an ASP program) consists of a set of rules. So, we define what a rule is.

Definition 14 (Rule)**.** A *rule* r is an expression:

$$a_0 \leftarrow b_1 \wedge \ldots \wedge b_k \wedge not\ b_{k+1} \wedge \ldots \wedge not\ b_m \;\; k \geq 0, m \geq 0, m \geq k$$

where $a_0, b_1, \ldots, b_m$ are atoms, a_0 is the head of the *rule* (denoted by $H(r)$) and $b_1, \ldots, b_m$ the body of the *rule* (denoted by $B(r)$).

If $H(r) = \bot$ then r is a *constraint*.

The positive body of r is $B(r)^+ = \{b_1, \ldots, b_k\}$ and $B(r)^- = \{b_{k+1}, \ldots, b_m\}$ is the negative body of r.

Definition 15 (Logic Program)**.** A *logic program* is a countable set of rules. If all rules are positive rules, then it is a *positive logic program*.

2.2.2 Semantics

Here we present the meaning of ASP programs defined by using the language previously defined and using some references. These references include the document of the original authors of answer set semantics [GL91] where it mentions the syntax, introduces stable models and also gives some examples. We also make reference to [Bar01] where it is focused the "language of logic programming with answer set semantics and its application to knowledge representation, reasoning and declarative problem solving", where it is possible to consult more aspects of this topic. We also make reference to other sources from where we adopt some concepts as [Alv11, MT99].

First, we define some concepts that help define what interpretation in this context is.

Definition 16 (Herbrand Universe)**.** Let P be a logic program. The set of all ground terms appearing in P is called the *Herbrand universe* of P.

Definition 17 (Herbrand Base)**.** Let P be a logic program. The *Herbrand base* of P is the set containing all ground atoms that can be constructed from the predicates in P and the terms in the *Herbrand universe*.

Definition 18 (Herbrand Interpretation)**.** [1] A *Herbrand interpretation* of a logic program P is any subset I of the *Herbrand base* of P.

We also present the semantics of a logic program P, translating it into a sentence:

Definition 19. Let P be a logic program and r a rule. The translation π, where ω is the vector of the free variables of r, is defined as:

- $\pi(r) = \forall_\omega(b_1 \wedge \cdots \wedge b_k \wedge not\ b_{k+1} \wedge \cdots \wedge not\ b_n \supset a_0)$;
- $\pi(P) = \{\pi(r) | r \in P\}$.

[1] Each *Herbrand interpretation* corresponds to a first-order model, that is, to a $\langle \mathbf{D}, \mathbf{I} \rangle$ pair. In particular, given a *Herbrand interpretation* I, the corresponding first-order model is $\langle \mathbf{D}, \mathbf{I} \rangle$ where $\mathbf{D}$ is the Herbrand universe of P and for every constant symbol c, every function symbol f and every predicate symbol p of arity n, $\mathbf{I}(c) = c$, $\mathbf{I}(f) = \emptyset$, $\mathbf{I}(p) = \{\langle c_1, \ldots, c_n \rangle \mid p(c_1, \ldots, c_n) \in I\}$.

Definition 20 (Herbrand Model)**.** A *Herbrand model* of a logic program P is a *Herbrand interpretaton* I of P such that $I \models \pi(P)$.

A *Herbrand model* I is minimal if no proper subset of I is a *Herbrand model* of P. That is, if there does not exist I' such that I' is a *Herbrand model* of P and $I' \subset I$.

A *Herbrand model* I is the *least model* of P if for all I', I' is a model of P implies $I \subseteq I'$.

Proposition 1. *Let P be a logic program. If P is a positive program and it has a model, then P has the least model.*

Definition 21 (Grounding)**.** Let r be a rule of the logic program P. The grounding of r in P, denoted by $ground(r)$, is the set of all rules obtained from r by all possible substitution of elements of Herbrand universe, for the variables in r. For any logic program P, grounding is defined as:

$$ground(P) = \bigcup_{r \in P} ground(r)$$

Definition 22 (Reduct)**.** Let P be a ground logic program. For any *Herbrand interpretation* of P, let P^I be the program obtained from P by deleting:

- Each rule that has a literal $not\ e$ in its body with $e \in I$;
- All literals of the form $not\ e$ in the bodies of the remaining rules.

Definition 23 (Stable Model/Answer Set)**.** Let P be a logic program. A Herbrand interpretation I is a *stable model* of P if I coincides with the least model of $ground(P)^I$.

If a rule of a logic program represents a constraint, then it has the effect of pruning those answer sets that match the body of the rule, changing the semantics of a program.

Definition 24 (Support)**.** [Alv11] Given a model I for a ground program P, we say that a ground atom A is supported by P in I if there exists a rule r of $ground(P)$ such that $H(r) = A$ and $B(r) \subseteq I$.

Proposition 2. *[MT99] Every stable model of P is a minimal model of P and a supported model of P.*

2.3 Databases

This section is divided in three aspects: the basic concepts, the ICs and the repairs. The basic concepts subsection clarifies the concepts about databases that are essential to understand the focus of the document. The subsection about integrity constraints introduces the various constraint types, clarifies their purpose and defines their semantics using first-order logic, but treating null values the same way as ordinary ones. The database repairs subsection explains some basic concepts in order to understand what is a repair and a minimal repair.

Throughout our definitions we assume that $\bar{x}$ and $\bar{y}$ are sequences of pairwise distinct variables with appropriate arity, such that x_i denotes the i^{th} component of $\bar{x}$.

2.3.1 Database concepts

With respect to database structure, in this document we focus on a specific data model that is the relational model which was introduced first by E. Codd in [Cod70]. Since then many database books focus on the use of that model and we refer these to define the main concepts of databases and also of the relational model like [SKS10] and [CMR09].

We start by introducing the domain concept. Each domain specify a set of values and it can include a value that represents the absence of information: the null value. An *attribute* is composed by a sequence of these values and each *attribute* has a corresponding domain. A set of *attributes* compose an entity, that is uniquely represented by those and each *attribute* represents a property of that entity. Each entity is also called a *tuple*.

One example is mentioned by [SKS10], regarding null values, where we have an attribute that represents a phone number in an instructor table. It may be the case that the instructor does not have any phone number. In this case, it is correct to use a null value because the value is unknown or does not exist. These values also complicate relational operations, namely arithmetic operations or even aggregates.

A *table* is a structure that groups a set of tuples that represent the same object/concept type. All *tuples* of a given *table* have the same properties, that is, the same *attributes*, with the same domains. Each *table* is organized into rows and columns, where each row represents a *tuple* and each column represents an *attribute*. Each row has a corresponding row identifier (rowid) which is number that is unique for each tuple of the database and that cannot be a null value.

In order to better understand these concepts, here is an example:

Example 4 (Table Composition Example)**.** A table about departments [SKS10]:

$dept\_name$	$building$	$budget$
$Comp.Sci.$	$Taylor$	100000
$Biology$	$Watson$	90000
$Elec.Eng.$	$Taylor$	85000
$Music$	$null$	80000
$Finance$	$Painter$	120000
$History$	$Painter$	50000
$Physics$	$Watson$	70000

Table 2.2: Departments table

On this example, each column of the table is an attribute, in this case, $dept\_name$, $building$ and $budget$ are attributes and each row is a tuple, like $\langle Elec.Eng., Taylor, 85000\rangle$, for example. The domain of $dept\_name$ and $building$ is a string and the domain of $budget$ is a number. We can also observe that we do not know what is the building of the Music department, so we represent it by using a null value.

Now, that we already introduced the concepts involved in the structure of a database,

as the tuples, the tables, and so on, and how these tables interact, we can define what a database is.

A database is an "integrated collection of data" [LM98]. It is used to represent "things" (entities) that are of interest to some organization and also specifies how they are related to each other. Therefore, a database is a structure that can store information about multiple types of entities, their attributes and relationships between those. It is a collection of tables and their relationships. Also, if we take a snapshot of the data in the database at a given instant in time, it is called a database instance. We can represent it using ground first-order atoms.

Definition 25 (Database)**.** A table is a tuple of the form $\langle N, \langle A_1, \ldots, A_n \rangle, R \rangle$, where N is a predicate symbol of arity n, $A_1, \ldots, A_n$ are pairwise different attributes names and $R \subseteq \mathbf{C}^n$. A database D is a set of tables.

Definition 26 (Database Instance Mapping)**.** [BBB03] A database instance D can be represented as a finite collection of ground atoms, denoted by $M(D)$:

$$M(D) = \{N(c_1, \ldots, c_n) | \langle N, A, R \rangle \in D \wedge \langle c_1, \ldots, c_n \rangle \in R\}$$

From this point forward, when we mention a database D, we consider that $D = M(D)$.

Also, in order to obtain useful information about data in a database, we query it. A query to a database is a statement requesting the retrieval of information [SKS10]. The standardized language used to formulate these queries is called Structured Query Language (SQL). It is "composed of commands that enable users to create database and table structures, perform various types of data manipulation and data administration, and query the database to extract useful information" [CMR09].

In this context, we denote the set of tables' names of a database D by $\mathcal{N}_D$ and use $\mathcal{A}_{D,N}$ to denote the sequence of attributes' names of a table N in a database D.

2.3.2 Integrity constraints

ICs form part of a database, capturing the real-world relationships among tables and data of that database. They are normally expressed in the form of rules.

These ICs can be of several types, for instance:

- Keys, unique constraints and functional dependencies: keys are used with the goal of uniquely identify tuples by means of a set of attributes, so that no two tuples are allowed to have the same values on all attributes of the key set. These type of keys are called candidate keys and can be expressed with unique constraints. The key chosen to identify each tuple in a table, that does not have unnecessary attributes, is the primary key [CMR09]. However, unique constraints do not forbid null values as opposed to primary keys.

A foreign key is a set of attributes of a table that refers a candidate key of another table. It guarantees that for all tuples in the first table there must be one tuple in the second table having the same values on all referred attributes.

Keys are special cases of functional dependencies. With functional dependencies, we can determine a set of attributes B by knowing a set of attributes A for a table R. The difference is that with keys we determine ALL the attributes of the table, while with functional dependencies, we just determine some of them;

- Inclusion Dependencies: an inclusion dependency is a set of attributes of a table that refers a set of attributes of the same or another table. They are similar to foreign keys, the only difference is that the referred set of attributes does not have to be a candidate key of that table;
- Denial constraints: we use them to prevent a property to hold in a database. They can be very general but here we just consider that they are divided into domain constraints, check constraints and not-null constraints.
  - Domain constraints: every attribute must have a domain of possible values. That is, when we declare the domain of an attribute, it acts as a constraint, limiting the values that that attribute can have;
  - Check constraints: permit to restrict the values of the attributes by using conditions;
  - Not-Null constraint: the domain of the attribute cannot include the null value.

Next, we specify some definitions to then formally represent the IC types. The following auxiliary definitions [2] are adapted from [Alv11].

Definition 27 (Attribute Index Function)**.** Let D be a database and $\langle N, \langle A_1, \ldots, A_n \rangle, R \rangle \in D$. # is defined as follows:

- For any attribute name A, $\#_A^N$:

$$\#_A^N = \begin{cases} \textit{undefined} & \textit{if } \forall_{i \in \{1,\ldots,n\}} A \neq A_i \\ i & \textit{if} A = A_i \end{cases}$$

- For any set of names of attributes X, let $\#_X^N$:

$$\#_X^N = \{\#_A^N | A \in X\}$$

Definition 28 (Arity Function)**.** Let N be a table name and n the arity of that table name. $\mathcal{AR}$ returns the arity of the table with name N and it is defined as:

$$\mathcal{AR}(N) = n$$

[2] In the following definitions we refer to a table by its name, using the N notation.

Keys, unique constraints and functional dependencies These IC types are similar because they have to check every tuples of the corresponding table in order to determine if there is a inconsistency. Also, they define some uniqueness for a sequence of attribute values that cannot be violated.

First, each primary key is a pair $\langle N, A\rangle^{pk}$ where N is a table name and A is a set of attributes that form the key constraint of that table.

Definition 29 (Semantics of Primary Keys)**.** Let $c = \langle N, A\rangle^{pk}$ be a primary key and $n = \mathcal{AR}(N)$. The semantics of c, denoted by $\beta_\sigma(c)$, is defined using the first-order formula:

$$\forall_{\bar{x},\bar{y}} \neg \left[N(\bar{x}) \wedge N(\bar{y}) \wedge \bigwedge_{i \in \#_A^N} x_i = y_i \wedge \bigvee_{j \in \{1,2,3,\ldots,n\} \setminus \#_A^N} x_j \neq y_j \right]$$

We also have unique constraints that, as primary keys, do not allow the database to have two tuples that have the same values on all attributes that form the unique set. However, we can have multiple unique constraints on a single table but we cannot have more than one primary key. Each unique constraint is a pair $\langle N, A\rangle^{uc}$ where N is a table name and A is a set of attributes that form the constraint of that table.

Definition 30 (Semantics of Unique Constraints)**.** Let $c = \langle N, A\rangle^{uc}$ be a unique constraint and $n = \mathcal{AR}(N)$. The semantics of c, denoted by $\beta_\sigma(c)$, is defined using the first-order formula:

$$\forall_{\bar{x},\bar{y}} \neg \left[N(\bar{x}) \wedge N(\bar{y}) \wedge \bigwedge_{i \in \#_A^N} x_i = y_i \wedge \bigvee_{j \in \{1,2,3,\ldots,n\} \setminus \#_A^N} x_j \neq y_j \right]$$

Each functional dependency is a tuple $\langle N, A, B\rangle^{fd}$ where N is a table name and two sets of attributes A and B from that table form the constraint.

Definition 31 (Semantics of Functional Dependencies)**.** Let $c = \langle N, A, B\rangle^{fd}$ be a functional dependency. The semantics of c, denoted by $\beta_\sigma(c)$, is defined using the first-order formula:

$$\forall_{\bar{x},\bar{y}} \neg \left[N(\bar{x}) \wedge N(\bar{y}) \wedge \bigwedge_{i \in \#_A^N} x_i = y_i \wedge \bigvee_{j \in \#_B^N} x_j \neq y_j \right]$$

Referential constraints This group of ICs represents a set of ICs that are similar because they require that the values of a set of attributes of a table also appears in the set of attributes of another table.

Each foreign key is a tuple $\langle N_1, N_2, A, B\rangle^{fk}$, with two table names N_1 and N_2, two sequences of attribute names $A = \langle A_1, \ldots, A_k\rangle$ and $B = \langle B_1, \ldots, B_k\rangle$ from the table N_1 and N_2 respectively. The attributes in A are referring the attributes in B. B must form a candidate key of table N_2.

Definition 32 (Semantics of Foreign Keys)**.** Let $c = \langle N_1, N_2, A, B\rangle^{fk}$ be a foreign key, A and B two sequences of attribute names where $A = \langle A_1, \ldots, A_k\rangle$ and $B = \langle B_1, \ldots, B_k\rangle$. The semantics of c, denoted by $\beta_\sigma(c)$, is defined using the first-order formula:

$$\forall_{\bar{x}} \exists_{\bar{y}} \left[\neg N_1(\bar{x}) \vee \left(N_2(\bar{y}) \wedge \bigwedge_{i=1}^{k} x_{\#_{A_i}^{N_1}} = y_{\#_{B_i}^{N_2}} \right) \right]$$

Each inclusion dependency is a tuple $\langle N_1, N_2, A, B\rangle^{incd}$, composed of two table names N_1 and N_2, two sequences of attribute names $A = \langle A_1, \ldots, A_k\rangle$ and $B = \langle B_1, \ldots, B_k\rangle$ from the table N_1 and N_2 respectively. The attributes in A are referring the attributes in B. The different between these and foreign keys is that these simply generalize foreign keys by no longer requiring that B form a candidate key in N_2.

Definition 33 (Semantics of Inclusion Dependencies)**.** Let $c = \langle N_1, N_2, A, B\rangle^{incd}$ be an inclusion dependency, A and B two sequences of attribute names where $A = \langle A_1, \ldots, A_k\rangle$ and $B = \langle B_1, \ldots, B_k\rangle$. The semantics of c, denoted by $\beta_\sigma(c)$, is defined using the first-order formula:

$$\forall_{\bar{x}} \exists_{\bar{y}} \left[\neg N_1(\bar{x}) \vee \left(N_2(\bar{y}) \wedge \bigwedge_{i=1}^{k} x_{\#_{A_i}^{N_1}} = y_{\#_{B_i}^{N_2}} \right) \right]$$

Example 5 (Inclusion Dependencies)**.** Considering the database D that has a table P with the tuple $\langle a, b, c\rangle$ where the table P has the sequence of attributes $\langle x, y, z\rangle$, the table R has the sequence of attributes $\langle v, w\rangle$ and considering the inclusion dependency represented by:

$$\langle P, R, \langle y, z\rangle, \langle v, w\rangle\rangle^{incd}$$

D only is consistent with respect to the specified IC if the tuple $R(b, c)$ also exists in the database D, otherwise, it violates the constraint.

Denial constraints This group of ICs represents a set of ICs that are similar because they express conditions that limit the domain of the attributes.

Each check constraint is a tuple $\langle N, A, \theta, V\rangle^{ckc}$ composed of a table name N, an attribute A, a mathematical operator $\theta \in \{>, <, \geq, \leq, =, \neq\}$ and a value V of the domain of A. The operator θ with respect to the value V, indicates the condition that attribute A has to respect.

Definition 34 (Semantics of Check Constraints)**.** Let $c = \langle N, A, \theta, V\rangle^{ckc}$ be a check constraint. The semantics of c, denoted by $\beta_\sigma(c)$, is defined using the first-order formula:

$$\forall_{\bar{x}} \left[N(\bar{x}) \wedge x_{\#_A^N} \theta\ V \right]$$

Each domain constraint is a tuple $\langle N, A, D_A \rangle^{dc}$ composed of a table name N, an attribute A and a domain D_A, express by: $D_A = \{val_1, val_2, \ldots, val_k\}$.

Definition 35 (Semantics of Domain Constraints)**.** Let $c = \langle N, A, D_A \rangle^{dc}$ be a domain constraint. The semantics of c, denoted by $\beta_s(c)$, is defined using the first-order formula:

$$\forall_{\bar{x}} \neg \left[N(\bar{x}) \wedge \bigwedge_{val \in D_A} x_{\#_A^N} \neq val \right]$$

A more specific case of domain constraints are not-null constraints. These do not allow an attribute to have null values. Each not-null constraint is a tuple $\langle N, A \rangle^{nn}$ composed of a table name N and an attribute A.

Definition 36 (Semantics of Not-Null Constraints)**.** Let $c = \langle N, A \rangle^{nn}$ be a not-null constraint. The semantics of c, denoted by $\beta_s(c)$, is defined using the first-order formula:

$$\forall_{\bar{x}} \neg \left[N(\bar{x}) \wedge x_{\#_A^N} = null \right]$$

A consistent database state is when a set composed by all ICs are satisfied by the database [CMR09].

Definition 37 (Database Consistency)**.** Given a database instance D, represented as a set of atoms of the form $P(c_1, \ldots, c_n)$, and a set of ICs F, we say that D is consistent with F if $D \models F$.

ICs help to keep the database consistent, but they may be costly to test. Database modifications can cause violations of integrity and when new constraints are added it is difficult to enforce them in the database [SKS10].

There are a few ways to try to keep databases consistent regarding ICs. One of them consists in declaring the ICs together with the database declaration and then the DBMS is the responsible for keeping it consistent by rejecting the actions that result in a violation of the ICs. However, there is another important factor that has to be taken into account: the null values. The admission of an IC in presence of null values depends of the chosen semantics for these values. There is no general agreement on the semantics of null values because the null value can be interpreted in many ways: it can express the unknown, inapplicable or withheld value [BB06].

There are three main semantics described in SQL Standard [Mel03] for foreign key constraints, in order to specify when tuples with null values respect the ICs:

- Simple match;
- Partial match;

- Full match.

Details about these semantics are approached in Chapter 4.

2.3.3 Repairs

After we know that a database is inconsistent with respect to the ICs, we can move on to the repairing process. Repairing consists of modifying the tuples of the database in order to obtain consistency through insertion, deletion or attribute update operations. Each alternative instance that results from different set of operations and that is consistent with the ICs, is called a repair. A database instance can have multiple possible repairs.

Example 6 (Repairing Students Table Example)**.** Considering the functional dependency IC from [Ber11], where $Students$ is the table name and $\{StuNum, StuName\}$ is the set of attributes of that table:

$$\langle Students, \{StuNum\}, \{StuName\}\rangle^{fd}$$

and a database D that collects information about students:

StuNum	*StuName*
101	*john bell*
101	*joe logan*
104	*claire stevens*
107	*pat norton*

Table 2.3: Students table

The database presents an inconsistency. The possible repairs are:

StuNum	*StuName*
101	*john bell*
104	*claire stevens*
107	*pat norton*

Table 2.4: Repair 1 of Students table

StuNum	*StuName*
101	*joe logan*
104	*claire stevens*
107	*pat norton*

Table 2.5: Repair 2 of Students table

These two repairs are consistent regarding the IC and they minimally differ from the original database. However, how to measure the distance between instances in order to choose the minimal one? We have to define a method to measure the differences between two databases in order to determine which alternative repair is closer to the original database. We can choose different criteria to measure these differences.

Definition 38 (Databases Symmetric Difference)**.** Let D_1 and D_2 denote two databases. The *symmetric difference* between D_1 and D_2, denoted by $\Delta(D_1, D_2)$, is:

$$\Delta(D_1, D_2) = (D_1 \backslash D_2) \cup (D_2 \backslash D_1)$$

Definition 39 (Closeness Relation)**.** [Ber11] Let D denote a database, D_1 and D_2 denote two database instances with respect to D and $\Delta(D_1, D_2)$ denote the set of changes between them. We say that D_1 is at least as close to D as D_2 , denoted $D_1 \leq_D^v D_2$ iff:

1. if $v = c$, then $\Delta(D, D_1) \subseteq \Delta(D, D_2)$;
2. if $v = s$, then $|\Delta(D, D_1)| \leq |\Delta(D, D_2)|$.

Definition 40 (Repair)**.** [ABC99] Let D be a database and F a finite set of ICs. A repair of D is a database G that satisfies F, that is, $G \models F$.

After we know what is a repair and how to compare repairs considering their distances with respect to the original database, we can define what is a minimal repair.

Definition 41 (Minimal Repair)**.** [ABC99] Let D be a database, F a finite set of ICs and v a closeness relation. A minimal repair of D is a database G such that $G \leq_D^v$-minimal in the class of instances that satisfy F.

The minimal repairs are the ones in which we have interest, since we prefer to keep the new database instance as close as possible to the original one. For example, we do not want to clear the whole database when deleting a single tuple would solve the inconsistency, because we would lose a lot of information unnecessarily. Therefore, we always prefer a repair that changes the database the least as possible.

3

Related work

In the previous chapter we introduced the basic concepts associated with the composition of databases, the type of ICs we can find when checking databases' consistency, what are repairs (and minimal ones) and how to use them to solve inconsistency problems so, now we can present some approaches that also discuss this topic and propose some solutions. In this chapter, we highlight some options regarding the repairing process without including null values. We will cope with these values from the next chapter onward.

Now, regarding existing approaches, most of them just assume that null values do not appear in the database to be repaired. Some of them are:

1. Characterizing and computing semantically correct answers from databases with annotated logic and answer sets [BBB03]: This approach uses a disjunctive logic program with annotation arguments and it specifies repairs as minimal models of a theory written in Annotated Predicate Logic. They establish a one to one correspondence between the models of that program and the repairs. Annotations are constants that specify when a tuple should be deleted or not or if it should be kept, for example;

2. Answer sets for consistent query answering in inconsistent databases [ABC03]: This approach uses logic programming based on answer sets to retrieve consistent data from an inconsistent database when a query is posed to the database using the DLV system. They determine database repairs using a disjunctive logic program with exceptions, declaring persistent data and changes to restore consistency by using exceptions. They provide approximated answers, using well-founded interpretations and identify the cases where it provides an exact solution;

3. Integrity constraints: semantics and applications [GGGM97]: This approach does not provide an implementation to repair databases. It is focused in ICs, what they mean, how they are used to represent knowledge of the database, reasoning with these and also mentions applications that employ ICs (like cooperative answering, semantic query optimization and so on). They express these ICs by transforming it into a disjunctive logic program and in modal logic;

4. The Consistency Extractor System: Querying inconsistent databases using answer set programs [CB07]: This approach presents an application that uses ASP to compute answers to first-order queries to relational databases. In this application the user can specify the ICs to be imposed and the query to the database. It distinguishes from other approaches because it uses an optimization technique called magic sets that transforms the query program and the repair program into a new one that contains a subset of the original rules in the repair program, those that are relevant to evaluate;

5. Sampling the repairs of functional dependency violations under hard constraints [BIG10]: Here, the authors propose to repair databases focused on functional dependencies, because their violations are "common in practice". They propose a new class of repairs and present an algorithm that randomly samples from this space of repairs. They focus on finding a meaningful number of repairs that can be generated in an efficient way. They also provide an application where the user can define the ICs and also the tuples or attributes that cannot be changed during the repairing process;

6. Efficient approximation algorithms for repairing inconsistent databases [LB07]: Here the authors do not develop a program to obtain the repairs of a database but they consider the problem of repairing a database that is inconsistent by updating numerical values. They focus is on denial ICs and, besides they do not provide an implementation, they provide the algorithms to do the repairing. Also, they provide an efficient approximation algorithm to obtain database repairs;

7. On approximating optimum repairs for functional dependency violations [KL09]: The authors present an approximation algorithm to produce repairs as closer as possible to the optimum ones, similarly to the previous approach, focusing on functional dependencies only and by only updating attribute values. They do not provide an implementation but they present the algorithms to use in the proposed approach;

8. Optimized encodings for consistent query answering via ASP from different perspectives [MT10]: Here the authors provide an uniform solution that comes from different perspectives of several notions of CQA under a common core and provide some optimizations to identify the inefficient part of the computation of consistent answers. This common core is defined using ASP and the problem of CQA is

reduced to disjunctive Datalog programs. To prove that the optimizations are significant they do some experiments that evidence the effectiveness of the approach. One of these optimizations is, for example, about excluding ICs that have no effect on the specific query.

Besides we have found many more approaches that not contemplate null values, we just analyze in more detail the most interesting and similar approaches of them, because the major interest is about including null values in the repairing process and the approaches to deal with them.

But first, as an aside, it is important to discuss a topic that is going to be emphasized when mentioning other approaches: the type of the repairing method. Here, we chose to adopt the DR because it tries to actually repair the inconsistent database by minimally modifying it so that the resulting database satisfies the ICs. In this alternative, a repair is seen as a natural and alternative "clean" version of the original instance. We get rid of semantic violations while staying as close as possible to the database. This is the natural approach because the database really is modified in order to always be consistent with the ICs and it refers to the entire database, not portions of it. This approach is inspired by the fact that violations are commonly due to different ways to refer the same entity so, the repair matches those ways into a single one, computing a single repair with a minimum cost [Ber11]. However, there is another alternative, the CQA, that is used when restoring the consistency of a database may be difficult, impossible or undesirable. We may not have the permission to do that and we may not want to lose useful data. This motivates the use of CQA which is based on the idea that not all data participate in the violation of ICs and the database can give semantically correct answers to some queries. In that case, inconsistency is not a big problem. Here, the ICs are not constraints on the database but on the sets of answers to queries [Ber11]. The answers are the data that matches the query and satisfies the ICs, that is, there are present in every minimal repair when the database cannot retrieve the data that is consistent with the ICs. We do not get rid of the tuples in the original database, so we do not lose information. However, we do not see the states of the database because we are always working with the inconsistent database and inconsistencies are solved at query time [BB04] and is also limited with respect to the class of ICs and queries it can handle [CB07].

The first approach that is analyzed is [Alv11] that was already mentioned as the only approach that provides an application that allows to repair a database.

With respect to the theoretical part, it is presented a transformation of the problem into a logic program and also the soundness and completeness proofs, as in this document, showing that the results obtained from the logic program correspond to the proposed approach and so, the application that uses the program behaves as defined theoretically.

In the developed application, the user can insert ICs to be enforced in the specified

database. The possible repairs are constructed through insert and delete operations, giving to the user the option of limiting the number of allowed operations. It is also possible to provide an extra database that is used to insert tuples in the database to be repaired or insert those tuples manually.

An interesting option that is provided in the implementation of [Alv11] is a SQL converter that allows the user to specify the ICs in SQL and then the program is responsible of converting them to ASP. This allows the user to specify ICs that are not possible to express using the application interface, without having to know ASP syntax. Besides, it is also possible to the user to do changes in the generated ASP code for each defined IC. After that specification, it is possible to choose the minimality criteria. The available criteria used are under set inclusion and under cardinality of operations. Also, the application automatically determines which are the relevant relations that should be included in the repairing procedure but in order to avoid to proceed with wrong relations in the case of the application do not correctly determine them, the user has the option of correcting them in the application interface.

As mentioned earlier, the [Alv11] approach does not take null values into account: it is not assumed that they are present in the database, nor used when doing insertions. Therefore, when inserting new tuples, the unknown values are asked to the user that is responsible by providing the extra database from which these values are extracted.

Also about the application, are provided some interesting optimizations that should be taken into account for our approach, for example:

- Importing ICs: only import ICs that are related directly or indirectly with the relevant tables;
- Deletions/insertions optimization: generate only the number of delete/insert operations that correspond to the maximum number of allowed operations for each one of them;
- Forbidding deletions in relations and of specific tuples: forbid deletions and forbid the deletion of a specific tuple;
- Maximum number of operations: specify a maximum global number of operations;
- Projection of attributes: consider only the relevant attributes.

In conclusion, the approach of this dissertation is very similar with this approach when not treating null values. The application to develop in the context of this dissertation is very similar to this one (more detail in Chapter 6), having the same goal of repairing a database but this time, with null values.

In [ABC99], it is made a logical characterization of the notion of consistent query answers in a relational database that may violate the given ICs. It is mentioned that solving this problem is useful for some areas like data warehousing and database integration.

In this approach, Arenas et al. do not provide a practical implementation, just the formalization. Although they left the completeness proof incomplete, because it is only made for some type of ICs, this is one of the few approaches that, like in this document, make the soundness and completeness proofs.

It is also presented a definition for the concept of consistency and introduced a repair definition that is based on the definition of distance between instances.

With respect to the allowed operations for consistency restoration, the choices are about insertions and deletions and the minimal criteria considered is under set inclusion.

Their approach is focused in optimizing the process of answering queries using semantic knowledge regarding the domain that is contained in the ICs. However, we do not analyze this in more detail because it is not directly related to the goal of this dissertation.

Lastly, the approach is similar to ours in the sense of presenting theoretical proofs. However, we do not present any optimizations to the process of answering queries because we do not use CQA and we present a practical component, as opposed to them.

In [GZ03], they propose a framework for computing repairs and consistent answers. There, it is stated that, when integrating two databases, it is not easy to define a preference criteria about the information sources. For example, if we have two tuples from two different databases (the ones that we are integrating), and when together in the same database provoke a IC violation, then it is not easy to choose just one to be part of the database that results from the integration because it is subjective and does not guarantee the consistency of the integrated database.

The contributions are a technique based on the rewriting of ICs, introduction of repair constraints and the introduction of prioritized updates.

About the rewriting of ICs, that allows the derived program to be used to both generate repairs for the database and produce consistent answers.

The repair constraints contribution allows to specify which repairs are feasible.

Regarding prioritized updates, they give preference to some repairs with respect to others, which is an interesting approach. However, the introduction of prioritized updates increases the complexity of computing repairs and one of the big disadvantages is the fact that this approach is neither ready for quantified variables in ICs, nor for answers containing null values. Also, it is proved that each of the three techniques is sound and complete and that the technique is more general than other techniques.

The approach works for both CQA and DR, and its mentioned how to proceed for both cases, being one of the few approaches that uses DR.

About the similarities with our approach, we also did the same choices with respect to some parameters as the allowed operations (insertions and deletions) and minimality criteria (under set inclusion), for example, and we also allow updates, as opposed to them, and cardinality minimal criteria. However we do not present prioritized updates but we provide to the user the option of choosing answers that make use of null values, reducing the number of available answers.

In [SMG10] an extended argumentation framework is introduced, which provides a comprehensive and alternative way to identify, represent and resolve the conflicts between inconsistent tuples in a database. The main goal of the framework is not to just repair the database but instead, to understand the reasoning process behind the repairing by using a technique that allows to explain and justify the conclusion of the process, to identify the conflicts (the inconsistencies) and find the pros and cons for a given conclusion through arguments (an argument "represents a statement about the relationship between a set of tuples and a set of ICs") and counter-arguments (an counter- argument "represents an alternative consistency restoration procedure for a given argument").

In order to restore the consistency of the database, the tuples that are in conflict are found and the authors define four types of conflicts that allow to identify which inconsistencies must be resolved in order to restore the database consistency. Also, Santos et al. define a notion of priority between pairs of conflicting tuples in order to establish preferences among repairs.

This article presents a new approach on relating argumentation with data repairing, although the decision support system that could assist the user in consistency restoration process was left as future work.

Comparing with other approaches, the use of argumentation gives the advantage of, besides resolving conflicts between tuples, justify those conflicts thought the understanding of the reasoning process behind database reparation. With respect to our approach, the similarities are just about the repairing method (DR) (they state that it maximizes database quality, being also one of the few approaches found that chose to use DR) and some options, for example, the operations allowed. However, there are not more similarities because the focus of this approach is not the repairing of databases but to understand the conflicts.

As a conclusion, we can see that the current state of the art with respect to this topic, have some common characteristics like the use of CQA or DR, the use of ASP to solve the problems, the allowed operations to perform in the repairing process (mostly insertions and deletions) or the minimality criteria (mostly under set inclusion). Although there are some proposes to deal with the problem, some of them are not directly focus on restoring the database consistency using DR and others simply do not provide a practical implementation.

References	Repair Type	Operations	Minimality criteria	Nulls?	Implementation?
[Alv11]	DR	Insert and delete	Set inclusion and Cardinality	No	Yes
[ABC99]	CQA	Insert and delete	Set inclusion	No	No
[GZ03]	CQA, DR	Insert and delete	Set inclusion	No	No
[SMG10]	DR	Insert and delete	(none)	No	Yes

Table 3.1: Approaches without null values summary

4

Taking null values into account: approaches and repairs

At this point, we already have an overall notion about what has been done to solve the problem of repairing databases without null values. However, we cannot assume that the information stored is perfect, therefore we cannot exclude the importance of null values. Null values occur very often in databases because it is common to have information that is unknown. When checking IC satisfaction with the existence of null values without specifying a behavior adapted to them, the results can be undesired, since we are considering null values as if they were regular values, but they are not. They have different interpretations and each interpretation can be represented with a different constant. For example, a null value can be missing or unknown, then we can represent a null value that is missing with a specific constant and a null value that is unknown with a different constant, and in this way, we would know which null values are missing and which are unknown. However, as mentioned, here, we represent all the interpretations with a single constant, as in commercial DBMSs, because we do not want to ask the user about the meaning associated to each null value in order to adopt different approaches.

Now that the meaning and the importance of considering null values in databases was already introduced, and how these values change the focus of the repairing databases problem, it is time to start to consider them in the repairing process. We begin by mentioning the repairing approaches that consider null values in some way, so we can reuse some results of those approaches in our analysis and approach and then we proceed to the proposed approach for the repairing process. We start with IC satisfaction and then we talk about the allowed operations and minimality criteria.

4.1 Approaches with null values

We begin with the approach in [BB04]. In this approach it is considered the presence of null values and it is suggested a semantics that is the focus of the document.

The provided semantics consider that, as null values represent no information, they not raise any inconsistencies, and so when at least one attribute of the tuple that is being checked has a null value, the tuple is considered to be consistent with the IC that is being tested. That is, all the tuples that have at least one null value in its attributes, have the necessary condition to be considered as consistent. Also, about the null values representation in logic programming, they are specified as a special constant.

With respect to the repairing, the repairs can be built with insert and delete operations and they must have a minimal distance from the original database. The insertions can also be obtained by introducing null values which avoid to generate a lot of alternative repairs, which is an advantage.

In order to get the repairs, the repair program runs together with a query program (since it used CQA method), the stable models are built and then is possible to extract the repairs making use of annotation constants. However, this approach faces the problem of obtaining undesirable models that do not correspond to repairs because not all the atoms that are minimized are relevant.

The authors also mention the problem of research being strong in computing stable models but not in query answering. However, we do not have to deal with this problem because we use DR instead of CQA, avoiding the use of queries to get a consistent database.

One suggestion that is given is to use a consistency check to determine if the query can be answered directly from the database and, in that case, the repair program can be avoided, because build stable models is a source of complexity.

The similarities of this approach with ours include treating null values by providing a semantics and also the use of null values to do insertions. Also, we have the advantage of not having problems with the minimization and so we do not get incorrect repairs. We also do not have to concern about the mentioned problem of query answering and we already implement the optimization that was proposed.

The next approach is [BB06] and it presents an improvement with respect to the previous approach.

First, the authors discuss how to represent a null value (with different constants or with a single constant) and it is decided to use a single constant to be according with commercial DBMSs. Then, are present the existent null semantics (in addiction to the semantics presented in [BB04]): simple-match, partial-match and full-match approaches. However, it is just considered the simple-match semantics, with the justification of being the one that is implemented by commercial DBMSs and there is not giving much attention to other possible semantics besides some simple examples. Also, the semantics are just

applied to foreign keys and in this approach it is extended to all types of ICs.

Besides the chosen semantics, regarding the repair decisions, the authors chose to use insert and delete operations. It is also discussed the case of inserting null values to avoid using values of the domain and consequently preventing to generate an infinite number of repairs, which is an advantage.

With respect to the transformation to a logic program, they use annotation constants that are used as an extra attribute introduced in the database. These annotations express when a tuple should be deleted or inserted.

To finalize, their approach only considers some types of ICs as foreign keys and not-null constraints.

This approach is actually close to ours here because it is considered the presence of null values on the database, present the same null semantics and propose a transformation into logic programming to then get the repairs. Besides, the representation of null values with one constant is common to our representation, the allowed operations include insertion and deletion operations, as in our approach, and it is also used minimality under set inclusion, as we do. However, the authors do not provide a careful study of the semantics and that is why we try to extend this approach and do a more careful analysis.

In [ABC03], the approach consists on specifying database repairs using disjunctive logic programs with exceptions. This extends [CAB00] by addressing several issues, like the extension of the methodology to more general ICs and the use of weak constraints to capture database repairs based on a minimal number of changes. Weak constraints differ from the regular constraints, because they have the effect of keeping the answer sets that minimize the number of violated ICs, instead of removing every answer set that violates at least one IC.

With respect to the allowed operations, the authors consider insert and delete operations and the minimality is measured by the set of changes. It is also give preference to use null values for insertion purposes and the other values are used only when it is really needed.

The main different between this approach and the other ones is the possibility of specify more "soft" constraints (weak constraints), allowing to express preferences instead of obligations, as in typical constraints. However, in our approach the ICs are considered as obligations and consequently they are always respected and so the user always get the more accurate solutions to repair the database.

We can also analyze a more "real" application, as in [FPL+01]. This system allows the correction of information of census questionnaires that may be incomplete (represented by null values) and/or contain inconsistent information. Here it is presented a real situation where it is necessary to represent unknown information and then it is initiated the flaws correction process. The result of this process should minimally diverge from

the original. In this case, that should be done in order not to impact significantly the statistics.

Here are considered strong and weak constraints. The use of strong constraints discards all models which do not satisfy some of them. However, in real life, sometimes we get satisfied by an approximated solution, in which the constraints are satisfied as much as possible. This is possible by the use of weak constraints, like was mentioned previously. Therefore, a resulting stable model satisfies all strong constraints and minimizes the number of violated weak constraints according to some specified prioritisation.

In order to solve the problem of considering null values, they use two approaches. In one of the approaches, when a questionnaire fails the rules, initially is made a search by similar questionnaires that passed the rules (donors) and that match the maximum as possible the attributes' values of the failed one. Then it analyses the rules to determine the minimum number of non-matching attributes between the failed one and the donor such that the failed one now can pass the rules. The other approach, first determines the minimum number of attributes and then searches by donors.

With respect to allowed operations, it is only used attribute updates in order to keep the information as closer as possible to the original one with the goal of not changing the statistics.

Comparing this approach with ours, it differs because it is a real application but that also needs to do repairs for a specific goal instead of doing repairs to general purpose databases. It is a perfect example to demonstrate the need of considering null values in real databases and their importance when doing databases repairing, as we consider in this dissertation. About the semantics, the authors do not really propose a semantics to deal with null values but instead there are presented two methods for dealing with these values in the context of this specific problem. Also, they use weak constraints but it is not considered an important feature for our purpose, as mentioned before.

In [AM86], the null values are adopted due to its characteristics of generality and naturalness and as a easy way to represent information on a fragment of the real world. In this article, the focus is only on the study of the interaction between two classes of ICs: functional dependencies and constraints on null values.

When in the presence of null values, a decomposition of tables can be achieved where tables with null values can be turned into tables where the presence of null values is somehow minimized. About null values when checking functional dependencies, where X determines Y, tuples with null values in the sequence of attributes X cannot cause a violation of the dependency because, as mentioned, null values mean that no information is available about those attributes. On the other hand, if the sequence X for two tuples is equal, and at least one of the attributes of Y is null, it causes a violation. The authors also highlight the importance of considering null values by stating that databases without null values assume that the real world is represented faithfully and that the knowledge is complete, and they highlight the fact that databases are approximations of the real world.

The similarities with this approach and ours are about the concern of treating null values and to provide reasons to consider them in databases. Also, they provide some kind of semantics to deal with theses values, but only with respect to functional dependencies, as opposite with ours, that consider semantics for all types of ICs.

As a conclusion, we can state that the use of null values is a problem that many times is ignored when repairing databases but there are some authors that realize that is a very real problem, and that it is necessary to represent information in databases like it is in real world. So, besides the small number of approaches that consider, in some way, null values, there are some that really try to assign semantics to null values and adopt the repairing procedure to these values. However, or the criteria used are too permissive or there is not a practical implementation.

References	Repair Type	Operations	Minimality criteria	Nulls?	Implementation?
[BB06]	CQA	Insert and delete	Set inclusion	Yes	No
[BB04]	CQA	Insert and delete	Set inclusion	Yes	No (current)
[ABC03]	CQA	Update	Set inclusion	On insert	No
[FPL$^+$01]	(kind of) DR	Update	(kind of) Set inclusion	Yes	Yes
[AM86]	(none)	(none)	(none)	Yes	No

Table 4.1: Approaches with null values summary

4.2 Integrity constraints and null values

Some alternative semantics for null values are presented in the literature that specify how to deal with them. However, some of the approaches do not consider how to treat these values for each IC type in particular and there is no global consensus on a specific semantics.

We continue the investigation started in [BB06], by considering alternative semantics other than simple-match applied to examples where we can interpret null values in different ways and try to conclude which is the best semantics for each example.

In the previous section, we mentioned some approaches that also present some semantics to deal with these values, for example, in [FPL$^+$01], the null values are replaced by values from a tuple that is as close as possible to the tuple that is being analyzed and in [AM86] it is present a semantics to deal with functional dependencies, but both are more focused in the context where they are presented. Other semantics is mentioned in [BB04], where it was explicitly introduced a semantics that is very permissive with null values but that can be applied to general cases. There are also three semantics mentioned on literature, namely in SQL Standard [Mel03], that we give more attention. We chose these semantics because, besides they are explicit on SQL Standard, they also are specified for foreign keys but they are applicable (or adaptable) to the other considered ICs.

1. Permissive semantics: a null value, as it represents a lack of information, should

not be a cause for an inconsistency when it is present in any of the attributes of a tuple [BB04]. So, for each row, if at least one of the values of the row is a null value then, it is not necessary to check the other attributes violate any constraint;

2. Simple-match: "for each row *R1* of the referencing table, either at least one of the values of the referencing columns in *R1* shall be a null value, or the value of the each referencing column in *R1* shall be equal to the value of the corresponding referenced columns in some row of the referenced table" [Mel03]. It is implemented in all commercial DBMSs;

3. Partial-match: "for each row *R1* of the referencing table, there shall be some row *R2* of the referenced table that the value of each referencing column in *R1* is either null or is equal to the value of the corresponding referenced column in *R2*" [Mel03];

4. Full-match: "for each row *R1* of the referencing table, either the value of every referencing column in *R1* shall be a null value, or the value of every referencing column in *R1* shall not be null and there shall be some row *R2* of the referenced table such that the value of each referencing column in *R1* is equal to the value of the corresponding referenced column in *R2*" [Mel03].

In summary, the permissive semantics considers that an IC is satified if at least one of the attributes has a null value. Similarly to the simple-match semantics that considers an IC as satisfied if at least one of the referring attributes has a null value, in opposite to full-match that, in most cases, if at least one of the referring attributes has a null value, then the IC is not satisfied. With partial-match semantics, we need to check if every referring attribute that does not have a null value has a correspondence with the referred attribute.

In order to better understand these semantics, here is an example:

Example 7 (Semantics Introduction Example)**.** Consider that we have a $Students$ table with the attributes $\{StudentID, Name, Age, School, Course\}$, the $Courses$ table with the attributes $\{CourseID, Course, School, MaxStudents\}$, the foreign key IC

$$\langle Students, Courses, \langle School, Course\rangle, \langle School, Course\rangle\rangle^{fk}$$

the candidate key $\{Course, School\}$ of $Courses$ table and the database:

StudentID	*Name*	*Age*	*School*	*Course*
E1	*John*	*null*	*School A*	*Average Course*
E2	*Claire*	*20*	*null*	*Basic Course*
E3	*Francis*	*21*	*Some School*	*null*

CourseID	*Course*	*School*	*MaxStudents*
C2	*Advanced Course*	*Some School*	*10*

Table 4.2: Students with Courses table

The results for the semantics are the following:

- Permissive: The IC is respected by all the tuples. For example, the first tuple of $Students$ table does not have a correspondence with the $Courses$ table but the Age attribute has a null value, so the IC is satisfied.
- Simple-match: The first tuple does not respect the IC since the only attribute that has a null value is not a relevant one and the attributes that are relevant to verify, do not have corresponding values in $Courses$ table. The second tuple satisfies the IC since one of the relevant attributes has a null value. However, in this tuple, the other relevant attribute does not have a corresponding value in $Courses$ table nor a null value: it refers a course that does not exist (does not appear in $Courses$ table);
- Partial-match: The first tuple of $Students$ table does not respect the IC because none of the relevant attributes has a value that is equal to the corresponding attribute of the referenced table and none of them has a null value. The last tuple has a null value in one of its relevant attributes and the other relevant attribute has a corresponding value in $Courses$ table, so it is consistent with the IC. Considering the last tuple, we do not have any information about the course of the student but we already know the school the student is studying. By knowing the school, a part of our IC is accomplished. Also, in this case, since that school only has one course, it is easy to guess the course of the student;
- Full-match: The first tuple is not satisfied since the attributes do not have corresponding values in the $Courses$ table. In this tuple, the null value is not a problem since its attribute does not belong to the relevant ones. The second and third tuples do not satisfy the IC since they have null values in its relevant attributes.

Tuples	Permissive	Simple	Partial	Full
$Students(E1, John, null, School\ A, Average\ Course)$	✓	×	×	×
$Students(E2, Claire, 20, null, Basic\ Course)$	✓	✓	×	×
$Students(E3, Francis, 21, Some\ School, null)$	✓	✓	✓	×

Table 4.3: Semantics introduction example results

As we can see from the results of the example, we can conclude that the more permissive semantics is not very appropriate to this kind of scenario because the Age attribute is not related to the purpose of the IC, that is to relate students with their courses and schools. So, the purpose of the IC is not accomplished yet.

With respect to the simple-match, its simplicity can complicate the main goal of ICs in modeling the given context by stating that much more tuples that intuitively should, satisfy the ICs.

Regarding partial-match, it is semantics that fits better until now, for this example. The partial-match approach is closer to the approach taken by commercial DBMSs in the absence of $null$ values.

About full-match, and regarding that we are appealing to the existence of null values in databases, this is not what we expect from a semantics because many tuples would be removed only by the occurrence of null values. The full-match approach is more restrictive than the other approaches, in sense that none of the relevant attributes can have a null value, otherwise it represents an inconsistency (but not if ALL attributes have null values).

The next step is to see some more examples where it becomes easier to see when it is better to use each semantics. However, we do not include the semantics of [BB04] because it is the most permissive approach and we have enough of it with simple-match, that is actually similar to the [BB04] approach. The type of explanations of the next subsection with respect to simple-match sometimes "fit" to the [BB04] approach too.

4.2.1 Examples and constraint satisfaction discussion

There are situations where the way we want to interpret null values vary and this is one of the reasons why it is difficult to deal with these values. So it is important to examine multiple examples in order to understand which is the better semantics to use in each situation, which may include different IC types.

Keys, unique constraints and functional dependencies With respect to primary keys, we do not compare the different semantics, since the primary key constraints do not allow null values in any of the attributes that compose the key.

Relatively to unique constraints, they are very similar to primary keys. However they allow the existence of null values and so we can compare the semantics.

Example 8 (Students Table Unique Constraint Example)**.** Consider that $\{Name, Age\}$ form the unique constraint and considering the following table:

StudentID	*Name*	*Age*	*Course*
E1	*John C.*	*null*	*Average Course*
E2	*Claire L.*	*20*	*Basic Course*
E3	*John C.*	*null*	*Advanced Course*

Table 4.4: Students table

In this specific example, since both students have the same name, then they probably represent the same person, they have the same age, and it makes sense to consider that both tuples correspond to the same person and to state that there is an inconsistency.

Here, there is no inconsistency considering simple-match since there are two tuples with the same value for the *Name* and *Age* attributes that correspond to different *Course* values. However, as the *Age* attribute has a null value, the IC is satisfied. The interpretation that we expected is not accomplished by the simple-match semantics in this case because one of the relevant attributes has a null value. Partial-match would behave as simple-match, since when we find two tuples that have in common the values of the relevant attributes, even the null values, the null values of one of the tuples may be different of the null values of the other tuple, then it would be enough to consider that we do not have an inconsistency because the uniqueness is not violated.

However, if we consider full-match, both tuples are rejected since they have null values on the relevant attributes, achieving what is expected. With full-match, we never have two tuples that represent the same student when in presence of null values in this type of examples.

In the next example we just change the focus of the Example 8, by introducing the null values in other attribute, that is, from an attribute that represents ages of persons to other that represents names of persons. The results are the following:

Example 9 (Students Table Unique Constraint Example Modified)**.** Consider a different version of the initial *Students* table:

StudentID	*Name*	*Age*	*Course*
E1	*null*	*21*	*Average Course*
E2	*Claire L.*	*20*	*Basic Course*
E3	*null*	*21*	*Advanced Course*

Table 4.5: Students table (2)

In the example, we have two tuples that have the same value for the *Age* attribute but we do not know the *Name* value for both. In this specific example, since we just know the age of a person (and not the name), we cannot state that if two persons have the same age, they are the same person. So, we should consider that the two tuples represent different persons. As consequence, if we choose to use full-match we get an unexpected result because we would find an inconsistency in the example and remove

the responsible tuples as if they were representing the same student and, so in this case, we expect the tuples to represent different persons and remain in the database, as would be possible with use simple-match/partial-match.

Also, if we exclude one of the tuples that full-match, in this context, would interpret as representing the same student and consequently violating the unique constraint, we get the following results:

Example 10 (Students Table Unique Constraint Example Modified - Full-match)**.** Consider a different version of the initial $Students$ table:

StudentID	*Name*	*Age*	*Course*
E1	*null*	*21*	*Average Course*
E2	*Claire L.*	*20*	*Basic Course*

Table 4.6: Students table (3)

Here, the problem of the previous example considering that two students with the same age and unknown names are violating the unique constraint, it not present no more. However, as one tuple has a null value in the $Name$ attribute there is still an inconsistency without any specific reason, despite there are not two tuples that have the same values for the other attribute that compose the unique constraint.

So, regarding the first two examples, we can see that changing just the attribute where the null values appear, can provoke a big change on the interpretation of them and consequently the need to use a different semantics. It just depends on the context.

However, in the first example, just by having two attributes with the same value for the $Name$ attribute does not mean that they are the same person because there can be more than a person with the same name and the attribute Age probably would unveil it (if known) and consequently, we would not have an inconsistency.

In the third example, we can see that just by adding one more tuple, it can change the interpretation that we give to a null value in that specific database instance. That is, it affects the choose of the semantics.

The previous examples related with unique constraints can also be applied to functional dependencies as 8, for example. The only difference would be to, instead of consider that $\{Name, Age\}$ form an unique pair, we consider that $Name$ determines Age, and the justifications are similar.

We provide an additional example, with respect to functional dependencies, that is about a different domain:

Example 11 (Cities Table Functional Dependency Example)**.** Consider the functional dependency IC

$$\langle Cities, \{Country, City\}, \{PostalCode\}\rangle^{fd}$$

and the database:

Country	*City*	*PostalCode*
null	*Lisbon*	*0001*
null	*Lisbon*	*0002*

Table 4.7: Cities table

Here, we consider that we have information about countries, cities and their postal codes. In the table of the example, if we have unknown countries and they represent the same, since they have a city in common, then we should have an inconsistency because we have the same pair of values for both tuples that correspond to different postal code values.

Here, with simple-match/partial-match, the database is consistent regarding the IC, because there are null values in the relevant attributes of both tuples. However, considering full-match appears to be a better option because we are expecting the null values to represent the same country.

However, if we had other tuples where the country is also unknown but city names that appear in the table are all different, then these tuples should not represent an inconsistency and full-match would suggest to delete them. So, none of the semantics is good enough for this example.

Referential constraints About foreign keys, considering that we have the Example 7, including the null values. Since our goal is to connect the student to the information about her course and school in their corresponding table (so they are connected with a foreign key), if for the second tuple of $Students$ table, we consider that this tuple is satisfying the constraint, since we know its course, we know half of the information that we need and later we can also know a school that has that course and add it to our $Courses$ table. If we accept this kind of flexibility, the best approach is the simple-match semantics. If we adopt full-match, then we are excluding all the students which the information of the course or of the school is unknown which can lead to the removal of the student. However, we may also allow the flexibility of accepting the IC as consistent of simple-match if we have a null value in one of the referring attributes but we demand that the referring attributes, since they are known, to exist in the corresponding table so that they do not refer to schools or courses that we do not know any information. This last situations matches with the behavior of partial-match.

The situations of this example do not always occur and to show that we consider the following example:

Example 12 (Organizations' Locations Example)**.** Consider the $Organizations$ table with the attributes $\{OrgID, OrgName, City, Country\}$, the $Locations$ table with the attributes $\{LocationID, City, Country, PostalCode\}$ and the foreign key IC, where $\{City, Country\}$

forms a candidate key for $Location$ table:

$$\langle Organizations, Locations, \langle City, Country\rangle, \langle City, Country\rangle\rangle^{fk}$$

and the database:

$OrgID$	$OrgName$	$City$	$Country$
$O1$	$Happy\ Company$	$null$	$Magic\ Country$
$O2$	$Sad\ Company$	$null$	$Portugal$

$LocationID$	$City$	$Country$	$PostalCode$
$L1$	$Lisbon$	$Portugal$	0001

Table 4.8: Organizations and locations table

Suppose that we ask some organizations to fill questionnaires in order to obtain information from them and that we intend to build a map with the organizations in a specific country. If we consider that the organizations are presented in the previous $Organizations$ table, we cannot represent them in a map if the city is not specified.

In this case, we can avoid to have organizations where do not know where they are located and consequently we will have null values representing these situations. So, full-match seems to be a better approach to state what we are looking for. Simple-match, for example, would accept as consistent every tuple that has a null value in the $City$ attribute, that is the opposite of what we intend. So, we would prefer to use full-match semantics than simple-match.

However, we may not want to remove all the tuples where we do not know the city, since if we have the country, we know that the unknown city is one of the cities of that country in $Locations$ table. This may seem similar to simple-match, but it simple would accept the tuples as consistent even if there are not tuples with the correspondent city in $Locations$ table, unlike partial-match. So, in this case we should consider the partial-match approach.

With this previous example and also referring to the Example 7, we can state that just considering the purpose for which the database is used we can conclude to allow or not the null values and consequently, the semantics. In this case, an intermediate approach probably be preferred so that it can fit to different purposes without completing excluding or completely allowing null values.

Denial constraints Regarding check constraints, the Example 13 illustrates this situation as presented in [BB06]:

Example 13 (Employee Table Check Constraint Example)**.** Consider the check constraint IC

$$\langle Employee, Salary, >, 100\rangle^{ckc}$$

and the database:

ID	*Name*	*Salary*
32	*null*	*1000*
41	*Paul*	*null*

Table 4.9: Employees table

In this case, we only need to check if the attribute $Salary$ has a value that is bigger than 100. Most DBMSs accept the IC when the condition evaluates to true or unknown, that is the case for both tuples of the table. If we do not accept the null value in this type of ICs, we are assuming that, for example in the second tuple, the null value in $Salary$ attribute represents a value that is lower than 100. The justification is the following: the IC states that the value has to be bigger than 100 to be accepted, by rejecting it we are stating the opposite, that is, that the value is lower than 100. However the null value can be representing many values that are above the 100 value and consider that null values are accepted as consistent seems to be a more reasonable approach for this particular case, that is, we should use simple-match. This also corresponds to the partial-match approach because, in general, they accept null values as consistent.

Full-match would consider that the second tuple represents an inconsistency.

We can also consider other check constraint like

$$\langle Cities, Country, =, \text{“}Portugal\text{”}\rangle^{ckc}$$

and the table 4.7.

Here, simple-match/partial-match do not make as sense as before because here we are stating that we only accept tuples which country corresponds to $\text{“}Portugal\text{”}$ and if we accept null values as consistent, we are allowing many cities that clearly belong to other countries that are not “ $Portugal$ ”. Here, seems to be more reasonable to not accept null values as consistent, that is, to use full-match.

Another example that illustrates another type of situation:

Example 14 (Products Table Check Constraint Example)**.** Consider the check constraint IC

$$\langle Products, Price, >, 0\rangle^{ckc}$$

and the database:

ProductName	*Price*
Pen	*5*
Pencil	*null*

Table 4.10: Products table

In this example, if we consider that we never insert a negative price number, then every null value would represent a positive number and so they should always be accepted as consistent, that is, using simple-match or partial-match.

As we can see in the examples, the need to allow null values prevails and so, in this case, probably simple-match/partial-match would be more reasonable choices.

With respect to domain constraints, we do not have much to discuss since we only have to decide in which cases we want to use null values in the attributes' domain or not.

Finally, there is nothing to compare with respect to not-null constraints since they just show when to accept or not a null value in a specific attribute.

4.2.2 Semantics with null values

In the previous subsection, we analyzed some cases where slightly changes could change the semantics that seems more reasonable to apply. However, there were cases where none of the semantics seemed to be "good enough", that is, any semantics cover all the cases in a reasonable way.

Taking this into account, overall, we choose to use partial-match as a semantic for the null values. Partial-match does not just consider that a single null value is the motive for accepting any IC as consistent, nor to reject immediately an IC. It accepts null values as consistent but also checks if the others not-null values are consistent with the corresponding table and the IC. Also, it is the most preferred semantics in the examples of the previous subsection as in 7, 13 and 9.

There are ICs that do not have a direct application of partial-match but we adapt it in order to follow the same line of thinking that was used in the foreign keys example.

From now on we consider that $\bar{x}$ and $\bar{y}$ are sequences of pairwise distinct variables with appropriate arity, such that x_i denotes the i^{th} component of $\bar{x}$.

Keys, unique constraints and functional dependencies Relatively to primary keys, we just consider that a primary key can only be a non-null value or a set of non-null values. So, each attribute that compose a primary has to contain a non-null value. This is the approach taken by most DBMSs e.g. MySQL and PostgreSQL. These DBMSs specify that any key column is defined with a not-null constraint. That is, this IC is basically an unique constraint together with a not-null constraint over each of the attributes that compose the primary key.

For example, if we consider the Table Students of the Example 7, if $\{Name, Age\}$ forms the primary key, then an inconsistency arise due to the first tuple.

Definition 42 (Semantics With Null Values of Primary Keys)**.** Let $c = \langle N, A\rangle^{pk}$ be a primary key, N a table name and $n = \mathcal{AR}(N)$. The semantics considering null values for that primary key, $\beta_{sn}(c)$, is defined as:

$$\forall_{\bar{x},\bar{y}}\neg\left[N(\bar{x}) \wedge N(\bar{y}) \wedge \left(\bigvee_{i\in\#_A^N} x_i = null \vee \Big(\bigwedge_{i\in\#_A^N} x_i = y_i \wedge \bigvee_{j\in\{0,1,2,3,\ldots,n\}\setminus\#_A^N} x_j \neq y_j\Big)\right)\right]$$

With respect to unique constraints, when comparing the attributes in order to check if they match, we consider that any comparison with a null value should return false since any operation with a null value results in an unknown value, as stated in SQL standard [Mel03]. Since, we do not know when two null values correspond to the same regular value, then we consider that any comparison with a null value returns false. This is also applied when comparing two null values, which returns false, that is, two null values are considered to be different values, for the same reason.

As we can see in the Modified Example 8, both partial-match and simple-match have the expected result, then we adopt what was defined in [BB06] (simple-match approach) but adapted to unique constraints.

Definition 43 (Semantics With Null Values of Unique Constraints)**.** Let $c = \langle N, A\rangle^{uc}$ be a unique constraint, N a table name and $n = \mathcal{AR}(N)$. The semantics considering null values for that unique constraint, $\beta_{sn}(c)$, is defined as:

$$\forall_{\bar{x},\bar{y}}\neg\left[N(\bar{x}) \wedge N(\bar{y}) \wedge \Big(\bigwedge_{i\in\#_A^N} x_i = y_i \wedge x_i \neq null \wedge y_i \neq null\Big) \wedge \bigvee_{j\in\{0,1,2,3,\ldots,n\}\setminus\#_A^N} x_j \neq y_j\right]$$

With respect to functional dependencies, when comparing the attributes that determine the value of other attributes, we consider that any comparison with a null value returns false with the same motives mentioned for the previous IC and that also corresponds to use simple-match.

For example, in Example 11, by our approach, there is not an inconsistency since we have two tuples with the same values in the $\{Country, City\}$ pair that correspond to different postal code values but the null values in the $Country$ attribute are considered to be different since the cities may belong to different countries (and simply have city names in common). One of the tuples may be representing Lisbon city from Portugal and the other one may be representing Lisbon city from United States (Maine), also, there can be more cities named Lisbon around the world.

Definition 44 (Semantics With Null Values of Functional Dependencies)**.** Let $c = \langle N, A, B\rangle^{fd}$ be a functional dependency. The semantics considering null values for that functional dependency, $\beta_{sn}(c)$, is defined as:

$$\forall_{\bar{x},\bar{y}}\neg\left[N(\bar{x}) \wedge N(\bar{y}) \wedge \Big(\bigwedge_{i\in\#_A^N} x_i = y_i \wedge x_i \neq null \wedge y_i \neq null\Big) \wedge \bigvee_{j\in\#_B^N} x_j \neq y_j\right]$$

Referential constraints Here, we can refer to the example 12 to see an example where partial-match works well. In that case, the goal is to know the location of certain organizations and so, by having the information about the country where they are placed, we already have half the information. Also, we do not want to remove existent organizations just because we do not know in what specific city they are placed.

Definition 45 (Semantics With Null Values of Foreign Keys)**.** Let $c = \langle N_1, N_2, A, B\rangle^{fk}$ be a foreign key, $A = \langle A1, \ldots, A_k\rangle$ and $B = \langle B1, \ldots, B_k\rangle$. The semantics considering null values for that foreign key, $\beta_{sn}(c)$, is defined as:

$$\forall_{\bar{x}}\exists_{\bar{y}}\left[\neg N_1(\bar{x}) \vee \left[N_2(\bar{y}) \wedge \Big(\bigwedge_{i=1}^{k} x_{\#_{A_i}^{N_1}} = y_{\#_{B_i}^{N_2}} \vee x_{\#_{A_i}^{N_1}} = null\Big)\right]\right]$$

With respect to inclusion dependencies, we proceed as in foreign keys, using partial-match. We can state that that makes sense to use the same approach since they are very similar constraints. The only difference is that in foreign keys, we consider that the referred set of attributes has to be a candidate key of the correspondent table and in inclusion dependencies, it does not happen.

Definition 46 (Semantics With Null Values of Inclusion Dependencies)**.** Let $c = \langle N_1, N_2, A, B\rangle^{incd}$ be a foreign key, $A = \langle A1, \ldots, A_k\rangle$ and $B = \langle B1, \ldots, B_k\rangle$. The semantics considering null values for that inclusion dependency, $\beta_{sn}(c)$, is defined as:

$$\forall_{\bar{x}}\exists_{\bar{y}}\left[\neg N_1(\bar{x}) \vee \left[N_2(\bar{y}) \wedge \Big(\bigwedge_{i=1}^{k} x_{\#_{A_i}^{N_1}} = y_{\#_{B_i}^{N_2}} \vee x_{\#_{A_i}^{N_1}} = null\Big)\right]\right]$$

Denial constraints With respect to denial constraints, we consider that a null value, since it is an unknown value, can be considered as consistent, for check constraints. But why did we not proceed in that way for the previous ICs? We consider that, if the attribute has a null value, we do not know the value it may assume if the information was known, it could be a value that is consistent with the IC, so we allow it. Until now, could it be applicable for the previous ICs? Yes, it could. However, check constraints are based in the use of operators and if otherwise we assumed that the null value should be considered as a motive for an inconsistency then we would considering that the value is consistent with the opposite operation that we are checking. Also, the SQL Standard [Mel03] refers that "a table check constraint is satisfied if and only if the specified <search condition> is not false for any row of a table". This way, if the evaluation of the condition results in true or unknown (in the case of having null values), then the check constraint is satisfied. We can also confirm that "DBMSs will accept as consistent any state where the condition (the consequent) evaluates to true or unknown" [BB06], for example by using PostgreSQL, since any attribute that has a null value is accepted as consistent by any

check constraint on that attribute.

Definition 47 (Semantics With Null Values of Check Constraints)**.** Let $c = \langle N, A, \theta, V\rangle^{ckc}$ be a check constraint. The semantics considering null values for that check constraint, $\beta_{sn}(c)$, is defined as:

$$\forall_{\bar{x}} \neg \left[N(\bar{x}) \wedge \neg (x_{\#_A^N} \theta\ V \vee x_{\#_A^N} = null) \right]$$

For domain constraints, we consider that the domain of an attribute is composed of the specified domain and the null value too. If we do not consider null values in the domains of the attributes then we would only have not-null values in the attributes and we would obtain repairs that do not contain null values. Besides, we are providing an approach to deal with their presence, and we also explained previously why it is important. For exceptional cases where we do not allow their presence, then the user can specify some not-null constraints and avoid those values for some specific attributes.

Definition 48 (Semantics With Null Values of Domain Constraints)**.** Let $c = \langle N, A, D_A\rangle^{dc}$ be a domain constraint where *null* $\in D_A$. The semantics considering null values for that domain constraint, $\beta_{sn}(c)$, is defined as:

$$\forall_{\bar{x}} \neg \left[N(\bar{x}) \wedge \Big(\bigwedge_{val \in D_A} x_{\#_A^N} \neq val \Big) \right]$$

Finally, about not-null constraints, we just consider that an attribute where we apply a not-null constraint cannot contain a null value.

Definition 49 (Semantics With Null Values of Not-Null Constraints)**.** Let $c = \langle N, A\rangle^{nn}$ be a not-null constraint. The semantics considering null values for that not-null constraint, $\beta_{sn}(c)$, is defined as:

$$\forall_{\bar{x}} \neg \left[N(\bar{x}) \wedge x_{\#_A^N} = null \right]$$

4.3 Repairs

Now that is known how to identify the ICs violations over the data, it is time to know how it can be correct, in order to get a consistent database. This is achieved by repairs.

Definition 50 (Database Repair Problem)**.** Let D and D_E denote two databases and F denote a set of ICs, such that $D \not\models F$. A database repair problem is $\langle D, D_E, F\rangle$.

It is relevant to clarify that, in the above definition, D_E is a database that is the source of tuples to be used in insertions to repair the database D.

First, we should determine what are the operations that allow to lead the database into a consistent state.

4.3.1 Operations: Insertions, deletions and updates

In order to define a repair, we have to apply some operations to the database to cause some change that makes the database consistent with the ICs. The alternative operations are: deletions, insertions and updates.

Definition 51 (Change Operation)**.** Let D be a database, N a table name of D, each t denote a tuple and A denote an attribute name. A change operation is defined as one of the following:

- $i(N, t)$: insertion change operation;
- $d(N, t)$: delete change operation;
- $u(N, t, A)$: update change operation.

Each change operation, when applied to a database, changes that databases' data. In addiction, the insert change operation requires a tuple t that is inserted in a database D; the delete change operation needs a tuple t is the tuple that is deleted from a database D and the update change operation has a third argument that denotes the name of the attribute that is being updated with a null value with respect to the t tuple.

Definition 52 (Set of Changes)**.** Let $P = \langle D, D_E, F\rangle$ be a database repair problem, t is a tuple, N a table name where $N \in \mathcal{N}_D$, A an attribute name. The *set of all possible changes with regard to* P, denoted by $\Delta(P)$ is defined as follows:

$$\Delta(P) = \{\, d(N,t) \mid \langle N, A, R\rangle \in D \wedge t \in R \,\} \cup$$

$$\left\{\, u(N,t,A) \;\middle|\; \langle N, \langle A_1, \ldots, A_n\rangle, R\rangle \in D \wedge t \in R \wedge \exists_{i(i\in\{\,1,2,\ldots,n\,\}\wedge A = A_i)} \,\right\} \cup$$

$$\{\, i(N,t) \mid \langle N, A, R\rangle \in D_E \wedge t \in R \,\}$$

Definition 53 (Update Change Operation Aggregate)**.** Let $P = \langle D, D_E, F\rangle$ be a database repair problem, $\Delta \subseteq \Delta(P)$, $\langle N, \langle A_1, \ldots, A_n\rangle, R\rangle \in D$ and $t = \langle t_1, \ldots, t_n\rangle \in R$ a tuple. The *aggregation of unitary updates to* t is the tuple $u^{aggr}(\Delta, N, t) = \langle t'_1, \ldots, t'_n\rangle$ where for every $i \in \{\, 1, 2, \ldots, n\,\}$

$$t'_i = \begin{cases} null & if\ u(N,t,A_i) \in \Delta \\ t_i & otherwise \end{cases}$$

Definition 54 (Change Set)**.** Let $P = \langle D, D_E, F\rangle$ be a database repair problem. A *change set* is a set $\Delta \subseteq \Delta(P)$ such that for every table name $N \in \mathcal{N}_D$ and the tuple t, the following holds:

1. if $d(N,t) \in \Delta$, then $i(N,t) \notin \Delta$;

2. if $u(N,t,A) \in \Delta$, then $i(N,t) \notin \Delta$ and $d(N, u^{aggr}(\Delta,N,t)) \notin \Delta$.

With respect to insert change operation, when we have to insert new tuples, we have to deal with the problem of needing information that is not present in the database.

We could overcome this problem by only consider deletion of tuples, so it would not be necessary to find new values to do insertions. However, it would cause a big loss of information, which is not desirable in a database context. So, we chose to consider insertions and also deletion of tuples. This way, we can choose between repairs obtained by insertion and deletion of tuples, and choose the one that better respects the minimality criteria that will be specified in the next subsection. So, the database does not have a big increase in its size (because if we always repair a database by doing insertions we are constantly increasing the size of the database), growing with the number of repairs, nor a big decrease in its size (because if we always repair a database by doing deletions we are constantly decreasing the size of the database).

Regarding the updates, we also do not consider attributes updates because, we do not want to change values that are already stored in the database and that have a context given by the other tuples' attribute values, and when replaced by other values, the information may makes no more sense. Also, after updating a value, that combination of attribute values, can be violating another constraint.

Although we do not allow the update of attribute values to different values, because of the mentioned problems, we allow to update them to null values. This has the advantage of just "removing" some values instead of removing the whole tuple, for example, simultaneously avoid mixing old values with new ones and avoid the generate of an alternative update for each possible value of the domain.

Since we treat each attribute update as a single update operation, we need to determine the resulting tuple that corresponds to the original one with the changes of the set of single attribute update operations to that tuple.

Besides the above, we also need to merge the set of operations with the original database in order to get the resulting database that is consistent with the ICs.

Definition 55 (Database and Change Set Merging)**.** Let $P = \langle D, D_E, F\rangle$ be a database repair problem, N denote a table, t and t' denote tuples, Δ denote a change set and E the result of applying Δ to D denoted as $E = D \oplus \Delta$. Then

$$E = D \cup \{\, N(t) \mid i(N,t) \in \Delta \,\}$$

$$\cup \{\, N(t') \mid t' = u^{aggr}(\Delta,N,t) \wedge t' \neq t \,\}$$

$$\setminus \{\, N(t) \mid d(N,t) \in \Delta \,\}$$

$$\setminus \{\, N(t) \mid u^{aggr}(\Delta,N,t) \neq t \,\}$$

Definition 56 (Set of Change Sets)**.** The *set of change sets that achieve a repair*, denoted by $\Delta(P)_S$, consists of all change sets Δ for P such that $D \oplus \Delta \models F$.

Definition 57 (Repair)**.** [1] Let $P = \langle D, D_E, F \rangle$ denote a database repair problem. A repair G is defined as a database instance such as there is a $\Delta \in \Delta(P)_S$ that $G = D \oplus \Delta$.

4.3.2 Minimality

It was mentioned that the performed operations should be minimal but that minimality can be achieved considering different criteria. Each measurement criteria is determined based on its partial order and it used to compare the alternative repairs. This measure results in a distance between the possible repairs, that is, how these repairs differ.

In the context of this dissertation, the minimality criteria that are used are the under set inclusion, cardinality and weight criteria. These first two were chosen because they are very common and at least one of them is mentioned in each approach referred in the related work, when the topic of minimal criteria is introduced. The third one, the weight criteria, compensates the fact of other two criteria give the same weight to every operation, the same importance. That is, deleting a whole tuple would be as wanted as an update of a single attribute. This criteria is useful for this kind of cases, where we want to give preference for some operations.

We can start by defining what is an ordering criteria but first we define some auxiliary definitions.

Definition 58 (Weight Assignment Function)**.** A *weight assignment function* is a function that maps change operations to real numbers, denoted by $\omega(c)$ where c is a change operation.

Definition 59 (Balanced Weight Assignment Function)**.** Let D be a database, each t denote a tuple from a table N, $n = \mathcal{AR}(N)$ and A is an attribute name. ω_d denotes the balanced weight assignment function that:

- $\omega_d(i(N,t)) = 1$;
- $\omega_d(d(N,t)) = 1$;
- $\omega_d(u(N,t,A)) = \frac{1}{n}$.

Definition 60 (Ordering Type)**.** An ordering type is denoted by T where T $\in$ {D_c,D_s,D_ω} For the case of D_ω, ω denotes a weight assignment function.

[1] This concept of repair is now different from the previous one in the sense that it is being obtained by considering null values strategies. However, the same name is adopted for a matter of simplicity.

About the above definition, an ordering type is basically a criteria to order two or more database instances. The criteria that we consider are: cardinality criteria (D_c), subset criteria (D_s) and weight criteria (D_ω).

The under set inclusion criteria consists on choosing the repairs that, if some of its changes subsets are removed, then the repair does not respect the constraints anymore. As we want the repairs to be minimal, if a subset of the modifications of the repaired instance are unnecessary, there is no interest in maintaining that repair. We do not want a repair that is not as minimal as possible.

The cardinality criteria consists on choosing the repairs which the number of operations performed (insertions, deletions and updates) are minimal. So, if we want that the repair instance is as close as possible to the original instance, we want a instance that is obtained by the smallest set of operations, since each operation does a modification on the original instance, getting more "distance" from the original instance.

The weight criteria consists on assigning a weight to each operation type. This one is similar to a cardinality criteria that uses the weight value one, that is, we are considering that each operation type has an weight of one. With weight criteria, we can assign different weight values to different operation types, minimizing the sum of weights multiplied by the number of operations associated with that weight. So, we preferably want to get repairs where the operations that compose them have a lower weight value.

Definition 61 (Strict preorder)**.** Given a set S, a *strict preorder over* S is an irreflexive and transitive binary relation over S. Given a strict preorder $<$ over S, for any subset R of S, the set of *minimal elements of* R *with regard to* $<$ is:

$$\min(R, <) = \{\, a \in R \mid \neg\exists_{b \in R} b < a \,\}$$

Definition 62 (Closeness Relation)**.** Let P be a database repair problem. A *closeness relation* is a strict preorder on $\Delta(P)_S$.

Definition 63 (Instances Ordering)**.** Let $P = \langle D, D_E, F\rangle$ denote a database repair problem, T denote an ordering type, $\Delta \in \Delta(P)_S$ and $\Delta' \in \Delta(P)_S$. $\Delta <_P^T \Delta'$ iff one of the following cases hold:

1. T=D_c then $|\Delta| \leq |\Delta'|$;
2. T=D_s then $\Delta \subsetneq \Delta'$;
3. T=D_ω then $\Sigma_{s\in\Delta}\, \omega(s) \leq \Sigma_{s'\in\Delta'}\, \omega(s')$.

Regarding the previous definition, we allow two change sets with the same cardinality or sum of weights to be indistinguishable minimal because two change sets can have the same cardinality/sum of weights and be composed by different operations and so one cannot be easily preferred with respect to the other. This happens as opposed to sets

of operations where one change set only can be a subset of other when they refer to the same operations.

So, here is the minimal repair definition considering the previous concepts:

Definition 64 (T-Minimal Repair)**.** Let $P = \langle D, D_E, F\rangle$ denote a database repair problem and T an ordering type.

A *T-minimal repair with regard to* P is a database G such that for some $\Delta \in \Delta_S(P)$, $D \oplus \Delta = G$ and $\Delta \in \min(\Delta_S(P), <_P^T)$.

Now, here is an example that integrates the mentioned concepts in a simple way.

Example 15 (Students Repairing Example)**.** Let $P = \langle D, D_E, F\rangle$ be a repair problem where the database D is the one from Example 7 (just the first and the third tuple of the students table), the following foreign key IC:

$$\langle Students, Course, \langle School, Course\rangle, \langle School, Course\rangle\rangle^{fk}$$

the candidate key $\{Course, School\}$ for $Courses$ table, the database:

StudentID	*Name*	*Age*	*School*	*Course*
E1	*John*	*null*	*School A*	*Average Course*
E3	*Francis*	*21*	*Some School*	*null*

Table 4.11: Students with courses table

CourseID	*Course*	*School*	*MaxStudents*
C2	*Advanced Course*	*Some School*	*10*

Table 4.12: Courses table

And also the tuples of D_E:

CourseID	*Course*	*School*	*MaxStudents*
C3	*Average Course*	*School A*	*null*
C4	*Medium Course*	*null*	*null*
C5	*Basic Course*	*School B*	*35*

Table 4.13: Courses table

Also, we consider the cardinality criteria but if the chosen was the under-set inclusion, the result would be the same.

The database is inconsistent considering partial-match because there is no tuple in $Courses$ table for the tuple $Students(E1, John, null, School\ A, Average\ Course)$. This can be repaired by deleting the tuple of $Students$ table, by inserting a new tuple into $Courses$ table or by updating the $Course$ and $School$ attribute of the mentioned tuple to $null$. To

insert a new tuple it is necessary to know the maximum number of students associated to that course, but it is unknown, so it could be considered to use all the possible values of the domain. However, it is not necessary if we use a null value. So, the repairs are:

The first repair is:

ID	*Name*	*Age*	*School*	*Course*
E3	*Francis*	*21*	*Some School*	*null*

Table 4.14: Repair 1 - Students with courses table

ID	*Course*	*School*	*MaxStudents*
C2	*Advanced Course*	*Some School*	*10*

Table 4.15: Repair 1 - Courses table

The second repairs is:

ID	*Name*	*Age*	*School*	*Course*
E1	*John*	*null*	*School A*	*Average Course*
E3	*Francis*	*21*	*Some School*	*null*

Table 4.16: Repair 2 - Students with courses table

ID	*Course*	*School*	*MaxStudents*
C2	*Advanced Course*	*Some School*	*10*
C3	*Average Course*	*School A*	*null*

Table 4.17: Repair 2 - Courses table

Third repair is:

ID	*Name*	*Age*	*School*	*Course*
E1	*John*	*null*	*null*	*null*
E3	*Francis*	*21*	*Some School*	*null*

Table 4.18: Repair 3 - Students with courses table

ID	*Course*	*School*	*MaxStudents*
C2	*Advanced Course*	*Some School*	*10*

Table 4.19: Repair 3 - Courses table

The first repair corresponds to removing the problematic tuple, the second corresponds to the insertion of a new tuple where we are replacing an infinite number of repairs, each one using different domain values, by a null value, and the third repair corresponds of replacing the $School\ A$ value and the $Average\ Course$ value by $null$.

In the following chapters we focus on applying our approach, namely the transformation propose (Chapter 5) and the resulting application and corresponding features (Chapter 6).

5

Approaching the problem by using logic programming

Now that we discussed the alternative approaches and made the decisions, we need to compute repairs that are based on that theory. So now, we specify how to transform the rules associated with the decisions previous specified (ICs specification, deletion, insertion and update of tuples) into a logic program that should give us the answer sets that correspond to the repairs of the specified database.

About the notation of the null values, in ASP, only the terms that are ground and belong to the Herbrand Universe, are considered. So, we represent the null values by using a specific constant symbol that is used for whatever interpretation is given to that null value.

Just for remembering, we use the notation $\mathcal{N}_D$ to denote the set of tables' names of a database D and $\mathcal{A}_{D,N}$ to denote the set of attributes' names of a database D and table name N. We also use $\overline{x}$, $\overline{y}$ and $\overline{w}$ to represent sequences of distinct variables with a specific arity (being x_0, y_0 and w_0 the corresponding *RowId* of the tuple) and $\overline{c}$, $\overline{d}$ and $\overline{e}$ that represent a sequence of constants (being c_0, d_0 and e_0 the corresponding *RowId*).

5.1 Determining the repairs

In order to build the logic program we need to define the rules that allow the representation of the problem namely the tuples of the database or the allowed operations, for example. We use the following predicate symbols (where n represents arities that are

dependent from the table that is being used):

$$\{p_N/n, p\_keep_N/n, p\_new_N/n, p\_update\_single_N^A/1 | N \in \mathcal{N}_D \wedge A \in \mathcal{A}_{D,N}\} \cup$$

$$\{insert/1, delete/1, p\_update_N^A/1, update_N^A/1, n\_insert/1, n\_delete/1, n\_update_N^A/1\} \cup$$

$$\{updateAttrVal_N^A/2, update\_aggr_N/n | N \in \mathcal{N}_D \wedge A \in \mathcal{A}_{D,N}\} \cup$$

$$\{aux_{N_1,N_2}^{a_i,b_i}/n | N_1, N_2 \in \mathcal{N}_D \wedge a_i \in \mathcal{A}_{D,N_1} \wedge b_i \in \mathcal{A}_{D,N_2}\} \cup$$

$$\{aux_{N_1,N_2}^{\langle a_i,\ldots,a_k\rangle,\langle b_i,\ldots,b_k\rangle}/n | N_1, N_2 \in \mathcal{N}_D \wedge k \in \mathbb{N} \wedge \forall_{i=0}^k a_i \in \mathcal{A}_{D,N_1} \wedge \forall_{i=0}^k b_i \in \mathcal{A}_{D,N_2}\} \cup$$

$$\{p\_deleted_N/n, p\_inserted_N/n, p\_was\_update_N/n, p\_updated_N/n\}$$

$$\{dom_N^A/1\}$$

The first step of the transformation consists in getting the tuples of database tables and represent them as facts by using some predicates, specifying the corresponding table name and the values of the different attributes. We also represent the source of extra tuples database (the ones that can be inserted) by using a different predicate and the possible update to attributes with another predicate. Then, we generate all the possible combinations of delete, insert and update operations that can compose a repair and next we specify the constraints that prune the answer sets that do not satisfy the ICs. Those constraints are represented by different types of rules depending of the type of the represented constraint.

Definition 65 (Generating Tuples Facts)**.** Let $\mathcal{P} = \langle D, D_E, F\rangle$ denote a database repair problem, the tuples facts of the database D and of the database D_E, denoted by $\varphi_T(\mathcal{P})$, are:

1. For every atom $N(\overline{c})$ in $M(D)$, $\varphi(\mathcal{P})$ contains:

$$p_N(\overline{c}).$$

2. For every atom $N(\overline{c})$ in $M(D_E)$, $\varphi(\mathcal{P})$ contains:

$$p\_new_N(\overline{c}).$$

3. For every atom $N(\overline{c})$ in $M(D)$ and for every A in $\mathcal{A}_{D,N}$, $\varphi_T(\mathcal{P})$ contains:

$$p\_update\_single_N^A(c_0).$$

Definition 66 (Generating Rules)**.** Let $\mathcal{P} = \langle D, D_E, F\rangle$ denote a database repair problem, the generating rules for all the possible insertions, deletions and updates, denoted by $\varphi_G(\mathcal{P})$ are:

1. For every relation $N \in \mathcal{N}_D$:

   (a) $delete(x_0) \leftarrow p_N(\overline{x}) \wedge not\ n\_delete(x_0) \wedge not\ update(x_0).$

   (b) $n\_delete(x_0) \leftarrow p_N(\overline{x}) \wedge not\ delete(x_0).$

   (c) $p\_keep_N(\overline{x}) \leftarrow p_N(\overline{x}) \wedge not\ delete(x_0) \wedge not\ update(x_0).$

2. For every relation $N \in \mathcal{N}_{D_E}$:

   (a) $insert(x_0) \leftarrow p\_new_N(\overline{x}) \wedge not\ n\_insert(x_0).$

   (b) $n\_insert(x_0) \leftarrow p\_new_N(\overline{x}) \wedge not\ insert(x_0).$

   (c) $p\_keep_N(\overline{x}) \leftarrow p\_new_N(\overline{x}) \wedge insert(x_0) \wedge not\ delete(x_0) \wedge not\ update(x_0).$

3. For every relation $N \in \mathcal{N}_D$, where $n = \mathcal{AR}(N)$:

   (a) For each $i \in \{1, 2, \ldots, n\}$:

   i. $p\_update_N^{A_i}(x_0) \leftarrow p_N(\overline{x}) \wedge p\_update\_single_N^{A_i}(x_0) \wedge not\ n\_update_N^{A_i}(x_0).$

   ii. $n\_update_N^{A_i}(x_0) \leftarrow p_N(\overline{x}) \wedge p\_update\_single_N^{A_i}(x_0) \wedge not\ p\_update_N^{A_i}(x_0).$

   (b) $updateAttrVal_N^A(x_0, x_{\#_A^N}) \leftarrow p_N(\overline{x}) \wedge not\ p\_update_N^A(x_0).$

   (c) $updateAttrVal_N^A(x_0, null) \leftarrow p_N(\overline{x}) \wedge p\_update_N^A(x_0).$

   (d) $update\_aggr_N(x_0, \overline{y}) \leftarrow updateAttrVal_N^{A_0}(x_0, y_0) \wedge \ldots \wedge updateAttrVal_N^{A_n}(x_0, y_n) \wedge p\_update_N^{A_i}(x_0).$

   (e) $update(x_0) \leftarrow update\_aggr_N(x_0, \overline{y}).$

   (f) $p\_keep_N(\overline{y}) \leftarrow update\_aggr_N(x_0, \overline{y}) \wedge not\ delete(x_0).$

About the updates, it is relevant to note that each *p_update* predicate represents an update to null for a specific attribute of a specific tuple. Each updatable attribute is represented by a *p_update_single* predicate. If an attribute of a tuple is updated with that predicate then the value of that attribute becomes null, otherwise the original value is kept (this is represented with the *updateAttrVal* predicate). Also the *update_aggr* predicate represents the tuple's values after all the updates to its attributes.

Next, after getting the tuples and generating all possible operations that may compose the repair, we define all the constraint rules.

Definition 67 (Primary Key Rules)**.** Let $\delta_{pk} = \langle N, A \rangle$ denote a primary key and F denote a set of constraints. For every primary key constraint where $\delta_{pk} \in F$, for each $j \in \{0, 1, 2, ..., n\} \backslash \#_A^N$, $\varphi_{\delta_{pk}}(F)$, that represents the set of rules corresponding to δ_{pk}, contains:

$$\bot \leftarrow p\_keep_N(\overline{x}) \wedge p\_keep_N(\overline{y}) \wedge \bigwedge_{i \in \#_A^N} x_i = y_i \wedge x_j \neq y_j.$$

Also, for each $i \in \#_A^N$:

$$\bot \leftarrow p\_keep_N(\overline{x}), x_i = null.$$

Definition 68 (Unique Constraint with Simple Match Rules)**.** Let $\delta_{uc} = \langle N, A \rangle$ denote a unique constraint and F denote a set of constraints. For every unique constraint where $\delta_{uc} \in F$, for each $j \in \{0, 1, 2, ..., n\} \backslash \#_A^N$, $\varphi_{\delta_{uc}}(F)$, that represents the set of rules corresponding to δ_{uc}, contains:

$$\bot \leftarrow p\_keep_N(\overline{x}) \wedge p\_keep_N(\overline{y}) \wedge \bigwedge_{i \in \#_A^N} (x_i = y_i \wedge x_i \neq null \wedge y_i \neq null \wedge x_j \neq y_j).$$

Definition 69 (Functional Dependency with Simple Match Rules)**.** Let $\delta_{fd} = \langle N, A, B \rangle$ denote a functional dependency and F denote a set of constraints. For every functional dependency constraint where $\delta_{fd} \in F$, for each $j \in \#_B^N$, $\varphi_{\delta_{fd}}(F)$, that represents the set of rules corresponding to δ_{fd}, contains:

$$\bot \leftarrow p\_keep_N(\overline{x}) \wedge \ p\_keep_N(\overline{y}) \wedge \bigwedge_{i \in \#_A^N} (x_i = y_i \wedge x_i \neq null \wedge y_i \neq null \wedge x_j \neq y_j).$$

Definition 70 (Foreign Key with Partial Match Rules)**.** Let $\delta_{fk} = \langle N_1, N_2, A, B \rangle$ denote a foreign key, F denote a set of constraints, A and B two sequences of attribute names where $A = \langle A_1, \ldots, A_k \rangle$ and $B = \langle B_1, \ldots, B_k \rangle$ and i an index where $i \in \{\, 1, 2, \ldots, k \,\}$. For every foreign key constraint $\delta_{fk} \in F$, $\varphi_{\delta_{fk}}(F)$, that represents the set of rules corresponding to δ_{fk}, contains:

$$aux_{N_1,N_2}^{A_i,B_i}(\overline{x}, \overline{y}) \leftarrow x_{\#_{A_i}^{N_1}} = y_{\#_{B_i}^{N_2}}.$$

$$aux_{N_1,N_2}^{A_i,B_i}(\overline{x}, \overline{y}) \leftarrow x_{\#_{A_i}^{N_1}} = null.$$

$$aux_{N_1,N_2}^{A,B}(\overline{x}) \leftarrow p\_keep_{N_1}(\overline{x}) \wedge p\_keep_{N_2}(\overline{y}) \wedge \bigwedge_{i=1}^{k} aux_{N_1,N_2}^{A_i,B_i}(\overline{x}, \overline{y}).$$

$$\bot \leftarrow p\_keep_{N_1}(\overline{x}) \wedge not\ aux_{N_1,N_2}^{A,B}(\overline{x}).$$

Definition 71 (Inclusion Dependency with Partial Match Rules)**.** Let $\delta_{incd} = \langle N_1, N_2, A, B \rangle$ denote an inclusion dependency, F denote a set of constraints, A and B two sequences of attribute names where $A = \langle A_1, \ldots, A_k \rangle$ and $B = \langle B_1, \ldots, B_k \rangle$ and i an index where $i \in \{\, 1, 2, \ldots, k \,\}$. For every inclusion dependency constraint where $\delta_{incd} \in F$, $\varphi_{\delta_{incd}}(F)$, that represents the set of rules corresponding to δ_{incd}, contains:

$$aux_{N_1,N_2}^{A_i,B_i}(\overline{x}, \overline{y}) \leftarrow x_{\#_{A_i}^{N_1}} = y_{\#_{B_i}^{N_2}}.$$

$$aux_{N_1,N_2}^{A_i,B_i}(\overline{x}, \overline{y}) \leftarrow x_{\#_{A_i}^{N_1}} = null.$$

$$aux^{A,B}_{N_1,N_2}(\overline{x}) \leftarrow p\_keep_{N_1}(\overline{x}) \wedge p\_keep_{N_2}(\overline{y}) \wedge \bigwedge_{i=1}^{k} aux^{A_i,B_i}_{N_1,N_2}(\overline{x},\overline{y}).$$

$$\bot \leftarrow p\_keep_{N_1}(\overline{x}) \wedge not\ aux^{A,B}_{N_1,N_2}(\overline{x}).$$

Definition 72 (Check Constraint accepting Nulls Rules)**.** Let $\delta_{ckc} = \langle N, A, \theta, V\rangle$ denote a check constraint and F denote a set of constraints. For every check constraint where $\delta_{ckc} \in F$, $\varphi_{\delta_{ckc}}(F)$, that represents the set of rules corresponding to δ_{ckc}, contains:

$$aux^{A}_{N}(\overline{x}) \leftarrow x_{\#^{N}_{A}} \theta\ V.$$

$$aux^{A}_{N}(\overline{x}) \leftarrow x_{\#^{N}_{A}} = null.$$

$$\bot \leftarrow p\_keep_{N}(\overline{x}) \wedge not\ aux^{A}_{N}(\overline{x}).$$

Definition 73 (Domain Constraint including Nulls Rules)**.** Let $\delta_{dc} = \langle N, A, Dom\rangle$ denote a domain constraint and F denote a set of constraints, where $Dom = \{val_1, val_2, \ldots, val_n\}$. For every domain constraint where $\delta_{dc} \in F$, $\varphi_{\delta_{dc}}(F)$, that represents the set of rules corresponding to δ_{dc}, contains:

For each $val_i \in Dom$, we add:

$$dom^{A}_{N}(val_i).$$

Where A stands for the name of the attribute, and:

$$\bot \leftarrow p\_keep_{N}(\overline{x}) \wedge not\ dom^{A}_{N}(x_{\#^{N}_{A}}) \wedge x_{\#^{N}_{A}} \neq null.$$

Definition 74 (Not-Null Rules)**.** Let $\delta_{nn} = \langle N, A\rangle$ denote a not-null constraint and F denote a set of constraints. For every not-null constraint where $\delta_{nn} \in F$, $\varphi_{\delta_{nn}}(F)$, that represents the set of rules corresponding to δ_{nn}, contains:

$$\bot \leftarrow p\_keep_{N}(\overline{x}) \wedge x_{\#^{N}_{A}} = null.$$

Now that the answers that have the necessary conditions are filtered, we have to extract them.

Definition 75 (Transformation Function)**.** Let $\varphi(\mathcal{P})$ denote the transformation function of a database repair problem $\mathcal{P} = \langle D, D_E, F\rangle$ to a logic program. The logic program, denoted by $\varphi(\mathcal{P})$ is:

$$\varphi(P) = \varphi_T(P) \cup \varphi_G(P) \cup \bigcup_{\delta \in F} \varphi(\delta)$$

After getting all the answer sets, we have to extract the useful information from them in order to obtain the tuples that actually remain in the repaired database and the operations that were made to have them.

Definition 76 (Extracting Function)**.** Let $\mathcal{P} = \langle D, D_E, F \rangle$ be a database repair problem and let X be an answer set of $\varphi(\mathcal{P})$. We can extract the repair atoms using the function α defined as:

$$\alpha_N(X) = \{N(\bar{c}) \mid p\_keep_N(\bar{c}) \in X\}$$

$$\alpha(X) = \bigcup_{N \in \mathcal{N}_D} \alpha_N(X)$$

And we can obtain the modifications with respect to the original database with:

$$\Delta(X) = \{\, d(N,t) \mid delete(t_0) \in X \wedge p_N(t) \in X \,\}$$

$$\cup \{\, i(N,t) \mid insert(t_0) \in X \wedge p\_new_N(t) \in X \,\}$$

$$\cup \{\, u(N,t,A) \mid p\_update_N^A(t_0) \in X \wedge p_N(t) \in X \,\}$$

These modifications are useful to extract so that we know which operations we have to do in the database in order to make it consistent.

The next step is to define some theorems in order to specify the relation between repairs and answer sets obtained by the logic program. Their proofs are about the soundness and completeness properties, so that every answer set corresponds to a valid database repair and every database repair corresponds to an answer set, corresponding with the defined theory.

In order to prove that the atoms of each answer set are consistent with the theory (that they constitute a repair), we start by proving that the extracted tuples correspond with the available ones: original database's tuples; the set of tuples to be inserted or that result from updates from the original tuples. Then we ensure that they are consistent with the constraints.

Theorem 3 (Repair Soundness)**.** *Let $\mathcal{P} = \langle D, D_E, F \rangle$ be a database repair problem. For every answer set X of $\varphi(\mathcal{P})$, $\alpha(X)$ is a repair of D with respect to $\mathcal{P}$.*

Proof (sketch). Let $\mathcal{P} = \langle D, D_E, F \rangle$ be a database repair problem and X an answer set of $\varphi(\mathcal{P})$ such that $\alpha(X)$ is a repair of D with respect to $\mathcal{P}$.

In order to $\alpha(X)$ be a repair of D with respect to $\mathcal{P}$, by Definition 57, we need to prove that $\exists_{\Delta \in \Delta(P)_S} \alpha(X) = D \oplus \Delta$. Assume that Δ that corresponds to $\Delta(X)$.

In order to $\Delta \in \Delta(P)_S$ it must be a change set, by Definition 56, that with respect to Definition 54, means that Δ:

1. for every $d(N,t) \in \Delta$, then $i(N,t) \notin \Delta$;
2. for every $u(N,t,A) \in \Delta$, then $i(N,t) \notin \Delta$ and $d(N, u^{aggr}(\Delta,N,t)) \notin \Delta$.

For the first case, if we assume that $delete(t_0) \in X$, considering that t_0 is the zero-th attribute of a tuple from D, since t_0 uniquely identifies that tuple in the database, there are no insertion rules for that tuple (since the tuples to be inserted are from D_E and to be deleted from D), and if $insert(t_0) \in X$ then by Proposition 2 there has to be an insertion rule for the tuple, resulting in a conflict. Then, $insert(t_0) \notin X$, consequently $insert(t_0) \notin X \vee p\_new_N(t) \notin X$ and so $i(N,t) \notin \Delta$.

For the second case, if we assume that $u(N,t,A) \in \Delta$, then $p\_update_N^A(t_0) \in X \wedge p_N(t) \in X$, and as in the previous case, $i(N,t) \notin \Delta$. Let $t' = u^{aggr}(\Delta,N,t)$, then $t'_0 = t_0$. Assuming that $d(N,t') \in \Delta$, then $delete(t'_0) \in X$ and $delete(t_0) \in X$, that consequently, by support, $update(t_0) \notin X$. As $update(t_0) \notin X$, by support, $update\_aggr_N(t_0,t') \notin X$ and also (since the $updateAttrVal$ predicate covers all cases (the null and not null attributes) and so it belongs to X), by support, $\neg\exists_{A_i}\ p\_update_N^{A_i}(t_0) \in X$, resulting in a conflict. Then $d(N,t') \notin \Delta$.

Then, Δ is a change set.

We now prove that, regarding the change set Δ, $\alpha(X) = D \oplus \Delta$. By Definition 55, it is equivalent to $\alpha(X) = D'$ where:

$$D' = D \cup \{\, N(t) \mid i(N,t) \in \Delta \,\}$$

$$\cup \{\, N(t') \mid t' = u^{aggr}(\Delta,N,t) \wedge t' \neq t \,\}$$

$$\setminus \{\, N(t) \mid d(N,t) \in \Delta \,\}$$

$$\setminus \{\, N(t) \mid u^{aggr}(\Delta,N,t) \neq t \,\}$$

$\alpha(X) = D'$ is equivalent to $\alpha(X) \subset D' \wedge D' \subset \alpha(X)$, then we first prove that $\alpha(X) \subset D'$.

First, we take some $N(t) \in \alpha(X)$, that by Definition 75, means that $p\_keep_N(t) \in X$. In order to $p\_keep_N(t) \in X$, it can be generated by three possible rules, considering support of Definition 24. We treat each one by cases:

- Case 1: $p_N(t) \in X$ and $delete(t_0) \notin X$ and $update(t_0) \notin X$.

  As $p_N(t) \in X$ then $N(t) \in D$ by Definition 65. Also, as $delete(t_0) \notin X$, then there is no $d(N,t) \in \Delta$ which implies that the tuple is not removed from D'.

  Since $update(t_0) \notin X$, then by support, $update\_aggr_N(t_0,t') \notin X$, which implies that $\neg\exists_{A_i} p\_update_N^{A_i}(t_0) \in X$. Regarding Δ, since $\neg\exists_{A_i} p\_update_N^{A_i}(t_0) \in X$ then $u(N,t,A) \notin \Delta$ which by Definition 53 means that $u^{aggr}(\Delta,N,t) = t$. Then t is not updated.

  These conditions imply that $N(t) \in D'$.

- Case 2: $p\_new_N(t) \in X$ and $insert(t_0) \in X$ and $delete(t_0) \notin X$ and $update(t_0) \notin X$.

  First, $p\_new_N(t) \in X$ and $insert(t_0) \in X$ imply that $i(N,t) \in \Delta$, and so $N(t) \in \{\, N(t) \mid i(N,t) \in \Delta \,\}$.

  As $delete(t_0) \notin X$, then there is no $d(N,t) \in \Delta$ which implies that the tuple is not removed from D′.

  Since $update(t_0) \notin X$, then by support, $update\_aggr_N(t_0,t') \notin X$, which implies that $\neg\exists_{A_i} p\_update_N^{A_i}(t_0) \in X$. Regarding Δ, since $\neg\exists_{A_i} p\_update_N^{A_i}(t_0) \in X$ then $u(N,t,A) \notin \Delta$ which by Definition 53 means that $u^{aggr}(\Delta,N,t) = t$. Then t is not updated.

  These conditions imply that $N(t) \in D'$.

- Case 3: $update\_aggr_N(t_0,t') \in X$ and $delete(t_0) \notin X$.

  Since, $update\_aggr_N(t_0,t') \in X$ then $\exists_{A_i} p\_update_N^{A_i}(t_0) \in X$ by support, that means that the attribute A_i of t has been updated to null.

  If the updated attribute already had a null value then the tuple stays equal and it trivially corresponds to Case 1.

  Otherwise, $N(t) \in \{\, N(t') \mid t' = u^{aggr}(\Delta,N,t) \wedge t' \neq t \,\}$. Also, $delete(t_0) \notin X$, then there is no $d(N,t) \in \Delta$ which implies that the tuple is not removed from D′. As mentioned, $t' = u^{aggr}(\Delta,N,t) \wedge t' \neq t$, which implies that $N(t)$ is not removed. This implies that $N(t) \in D'$.

We now prove that $D' \subset \alpha(X)$. We take some $N(t) \in D'$, and prove that $p\_keep_N(t) \in X$. $N(t)$ can be included in D′ by three different alternatives. We treat each one by cases:

- Case 1: $N(t) \in D$

  By Definition 65, $N(t) \in D$ implies that $p_N(t) \in X$. Also, as we extracted $N(t)$ from D', it implies that $N(t) \notin \{\, N(t) \mid d(N,t) \in \Delta \,\}$, which implies that $d(N,t) \notin \Delta$. As $d(N,t) \notin \Delta$ then, $delete(t_0) \notin X \vee p_N(t) \notin X$. We previous stated that $p_N(t) \in X$. Both these statements, show a contradiction with respect to $p_N(t)$ and so we can state that $delete(t_0) \notin X$.

  Also, as we extracted $N(t)$ from D', it implies that $N(t) \notin \{\, N(t) \mid u^{aggr}(\Delta,N,t) \neq t \,\}$. That is, there is no $u(N,t,A) \in \Delta$ and so $p\_update_N^A(t_0) \notin X \vee p_N(t) \notin X$. We previous stated that $p_N(t) \in X$. Both these statements, show a contradiction with respect to $p_N(t)$ and so we can state that $p\_update_N^A(t_0) \notin X$. Also, as $p\_update_N^A(t_0) \notin X$, then by Definition 66, then $update\_aggr_N(t_0,t') \notin X$ and consequently $update(t_0) \notin X$. Then, as $p_N(t) \in X$, $delete(t_0) \notin X$ and $update(t_0) \notin X$, consequently, as X is a model, for all rule r if $X \models B(r)$, then it implies $X \models H(r)$, so $p\_keep_N(t) \in X$.

- Case 2: $N(t) \in \{\, N(t) \mid i(N,t) \in \Delta \,\}$

  Since $i(N,t) \in \Delta$ it implies that $insert(t_0) \in X \wedge p\_new_N(t) \in X$. Also, as we extracted $N(t)$ from D', it implies that $N(t) \notin \{\, N(t) \mid d(N,t) \in \Delta \,\}$, which implies

that $d(N,t) \notin \Delta$. As $d(N,t) \notin \Delta$, then $delete(t_0) \notin X \vee p_N(t) \notin X$. Assuming that $delete(t_0) \in X$. Then, by support, $p_N(t) \in X$ which is a contradiction. Then $delete(t_0) \notin X$.

Also, as we extracted $N(t)$ from D', it implies that $N(t) \notin \{\, N(t) \mid u^{aggr}(\Delta, N, t) \neq t \,\}$. That is, there is no $u(N,t,A) \in \Delta$. Even because $p\_new_N(t) \in X$, then it cannot be updated, consequently $update(t_0) \notin X$.

Then, as $insert(t_0) \in X$, $p\_new_N(t) \in X$, $delete(t_0) \notin X$ and $update(t_0) \notin X$, consequently, as X is a model, for all rule r if $X \models B(r)$, then it implies $X \models H(r)$, so $p\_keep_N(t) \in X$.

- Case 3: $N(t) \in \{\, N(t') \mid t' = u^{aggr}(\Delta, N, t) \wedge t' \neq t \,\}$

  Since $N(t) \in \{\, N(t') \mid t' = u^{aggr}(\Delta, N, t) \wedge t' \neq t \,\}$, it implies that $\exists_A u(N,t,A) \in \Delta$ by Definition 53, and this implies $p\_update^A_N(t_0) \in X \wedge p_N(t) \in X$.

  Also, as we extracted $N(t)$ from D', it implies that $N(t) \notin \{\, N(t) \mid d(N,t) \in \Delta \,\}$, which implies that $d(N,t) \notin \Delta$. As $d(N,t) \notin \Delta$ then, $delete(t_0) \notin X \vee p_N(t) \notin X$. Assuming that $delete(t_0) \in X$. Then, by support, $p_N(t) \in X$ which is a contradiction. Then $delete(t_0) \notin X$.

  Also, as we extracted $N(t)$ from D', it implies that $N(t) \notin \{\, N(t) \mid u^{aggr}(\Delta, N, t) \neq t \,\}$. Then, by Definition 53, $\exists_A u(N,t,A) \in \Delta$ and consequently $p\_update^A_N(t_0) \in X \wedge p_N(t) \in X$. As $\exists_A p\_update^A_N(t_0) \in X$ then $update\_aggr_N(t_0, t') \in X$.

  Then, as $delete(t_0) \notin X$ and $update\_aggr_N(t_0, t') \in X$, consequently, as X is a model, for all rule r if $X \models B(r)$, then it implies $X \models H(r)$, so $p\_keep_N(t) \in X$.

So, as $\alpha(X) \subset D'$ and $D' \subset \alpha(X)$ then $\alpha(X) = D'$.

In addiction to prove that $\Delta \in \Delta(P)_S$, we also need to show that $D \oplus \Delta \models F$, since $\alpha(X) = D \oplus \Delta$, then $\alpha(X) \models F$. So, we take some $\delta \in F$ and it has to respect one of the following cases:

1. δ is a primary key:

   For any two vectors of terms $\bar{c}$ and $\bar{d}$. The formula:

$$\neg\left[N(\bar{c}) \wedge N(\bar{d}) \wedge \left(\bigvee_{i \in \#^N_A} c_i = null \vee \Big(\bigwedge_{i \in \#^N_A} c_i = d_i \wedge \bigvee_{j \in \{0,1,2,3,\ldots,n\} \setminus \#^N_A} c_j \neq d_j\Big)\right)\right]$$

   is true iff:

$$\left[N(\bar{c}) \wedge N(\bar{d}) \wedge \left(\bigvee_{i \in \#^N_A} c_i = null \vee \Big(\bigwedge_{i \in \#^N_A} c_i = d_i \wedge \bigvee_{j \in \{0,1,2,3,\ldots,n\} \setminus \#^N_A} c_j \neq d_j\Big)\right)\right]$$

   is false.

   Suppose that $N(\bar{c})$ is true, $N(\bar{d})$ is true, then:

- if c_i for some $i \in \#_A^N$ is null, then X is not a model of the rules of Definition 67 and so we have a conflict;
- if c_i for all $i \in \#_A^N$ is not null, then if $\bigwedge_{i \in \#_A^N} c_i = d_i \wedge \bigvee_{j \in \{0,1,2,3,\ldots,n\}} c_j \neq d_j$ is true, a rule is violated by X, then X is not a model of the rules of Definition 67 and so we have a conflict.

For both cases, we have a conflict when the vector of terms corresponds to the expression, which means that that cannot exist an answer set X where two tuples correspond to the expression.

2. δ is an unique constraint:

   For any two vectors of terms $\bar{c}$ and $\bar{d}$. The formula:

$$\neg\left[N(\bar{c}) \wedge N(\bar{d}) \wedge \Big(\bigwedge_{i \in \#_A^N} c_i = d_i \wedge c_i \neq null \wedge d_i \neq null\Big) \wedge \bigvee_{j \in \{0,1,2,3,\ldots,n\} \setminus \#_A^N} c_j \neq d_j\right]$$

   is true iff:

$$\left[N(\bar{c}) \wedge N(\bar{d}) \wedge \Big(\bigwedge_{i \in \#_A^N} c_i = d_i \wedge c_i \neq null \wedge d_i \neq null\Big) \wedge \bigvee_{j \in \{0,1,2,3,\ldots,n\} \setminus \#_A^N} c_j \neq d_j\right]$$

   is false.

   Suppose that $N(\bar{c})$ is true, $N(\bar{d})$ is true, and $\bigwedge_{i \in \#_A^N} c_i = d_i \wedge c_i \neq null \wedge d_i \neq null$ is true then:

   - If $\bigvee_{j \in \{0,1,2,3,\ldots,n\} \setminus \#_A^N} c_j \neq d_j$ is true, a rule of Definition 68 is violated by X, then X is not a model of the rules and so we have a conflict.

   In that case, we have a conflict when the vector of terms corresponds to the expression, which means that that cannot exist an answer set X where two tuples correspond to the expression.

3. δ is a functional dependency:

   For any two vectors of terms $\bar{c}$ and $\bar{d}$.The formula:

$$\neg\left[N(\bar{c}) \wedge N(\bar{d}) \wedge \Big(\bigwedge_{i \in \#_A^N} c_i = d_i \wedge c_i \neq null \wedge d_i \neq null\Big) \wedge \bigvee_{j \in \#_B^N} c_j \neq d_j\right]$$

   is true iff:

$$\left[N(\bar{c}) \wedge N(\bar{d}) \wedge \Big(\bigwedge_{i \in \#_A^N} c_i = d_i \wedge c_i \neq null \wedge d_i \neq null\Big) \wedge \bigvee_{j \in \#_B^N} c_j \neq d_j\right]$$

   is false.

Suppose that $N(\overline{c})$ is true, $N(\overline{d})$ is true, and $\bigwedge_{i \in \#_A^N} c_i = d_i \wedge c_i \neq null \wedge d_i \neq null$ is true then:

- If $\bigvee_{j \in \#_B^N} c_j \neq d_j$ is true, a rule of Definition 69 is violated by X, then X is not a model of the rules and so we have a conflict.

We have a conflict when the vector of terms corresponds to the expression, which means that that cannot exist an answer set X where two tuples correspond to the expression.

4. δ is a foreign key:

   For any vector of terms $\overline{c}$, the formula:

$$\left[\neg N_1(\overline{c}) \vee \exists_{\overline{d}}\Big[N_2(\overline{d}) \wedge \Big(\bigwedge_{i=1}^{k} c_{\#_{A_i}^{N_1}} = d_{\#_{B_i}^{N_2}} \vee c_{\#_{A_i}^{N_1}} = null\Big)\Big]\right]$$

   is true, considering that there is a vector of terms $\overline{d}$. Regarding Definition 70, $aux_{N_1,N_2}^{A,B}(\overline{c})$ must be true so that the model is not removed. By support, there must be a $\overline{d}$ such that $p\_keep_N(\overline{d}) \in X$ and so all $aux_{N_1,N_2}^{A_i,B_i}(\overline{c},\overline{d}) \in X$.

   Suppose that $N_1(\overline{c})$ is true, that is, $\neg N_1(\overline{c})$ is false, then:

   - If $N_2(\overline{d}) \wedge \Big(\bigwedge_{i=1}^{k} c_{\#_{A_i}^{N_1}} = d_{\#_{B_i}^{N_2}} \vee c_{\#_{A_i}^{N_1}} = null\Big)$ is false, then a rule of Definition 70 is violated by X, then X is not a model of one of the rules and so we have a conflict.

   We have a conflict when the vector of terms does not correspond to the expression, which means that that cannot exist an answer set X that does not correspond to the expression.

5. δ is an inclusion dependency:

   For any vector of terms $\overline{c}$, the formula:

$$\left[\neg N_1(\overline{c}) \vee \exists_{\overline{d}}\Big[N_2(\overline{d}) \wedge \Big(\bigwedge_{i=1}^{k} c_{\#_{A_i}^{N_1}} = d_{\#_{B_i}^{N_2}} \vee c_{\#_{A_i}^{N_1}} = null\Big)\Big]\right]$$

   is true, considering that there is a vector of terms $\overline{d}$. Regarding Definition 71, $aux_{N_1,N_2}^{A,B}(\overline{c})$ must be true so that the model is not removed. By support, there must be a $\overline{d}$ such that $p\_keep_N(\overline{d}) \in X$ and so all $aux_{N_1,N_2}^{A_i,B_i}(\overline{c},\overline{d}) \in X$.

   Suppose that $N_1(\overline{c})$ is true, that is, $\neg N_1(\overline{c})$ is false, then:

   - If $N_2(\overline{d}) \wedge \Big(\bigwedge_{i=1}^{k} c_{\#_{A_i}^{N_1}} = d_{\#_{B_i}^{N_2}} \vee c_{\#_{A_i}^{N_1}} = null\Big)$ is false, then a rule of Definition 71 is violated by X, then X is not a model of one of the rules and so we have a conflict.

We have a conflict when the vector of terms does not correspond to the expression, which means that that cannot exist an answer set X that does not correspond to the expression.

6. δ is a check constraint:

   For any vector of terms $\bar{c}$.The formula:

$$\neg\left[N(\bar{c}) \wedge \neg(c_{\#_A^N}\theta\ V \vee c_{\#_A^N} = null)\right]$$

   is true iff:

$$\left[N(\bar{c}) \wedge \neg(c_{\#_A^N}\theta\ V \vee c_{\#_A^N} = null)\right]$$

   is false.

   Suppose that $N(\bar{c})$ is true then:

   - if $c_{\#_A^N}$ is not null, then X is not a model of one of the rules of Definition 72 but it is dependent of $c_{\#_A^N}$;
   - if $c_{\#_A^N}$ is null, then we still need to show $\neg c_{\#_A^N}\theta\ V$ is false. Assuming it is true, a rule of Definition 72 is violated by X, then X is not a model of the rules and so we have a conflict.

   We have a conflict when the vector of terms corresponds to the expression, which means that that cannot exist an answer set X where the tuples correspond to the expression.

7. δ is a domain constraint:

   For any vector of terms $\bar{c}$.The formula:

$$\neg\left[N(\bar{c}) \wedge \Big(\bigwedge_{val \in D_A} c_{\#_A^N} \neq val\Big)\right]$$

   is true iff:

$$\left[N(\bar{c}) \wedge \Big(\bigwedge_{val \in D_A} c_{\#_A^N} \neq val\Big)\right]$$

   is false.

   Suppose that $N(\bar{c})$ is true, then:

   - If $\bigwedge_{val \in D_A} c_{\#_A^N} \neq val$ is true, a rule of Definition 73 is violated by X, then X is not a model of one of the rules and so we have a conflict.

   We have a conflict when the vector of terms corresponds to the expression, which means that that cannot exist an answer set X where each tuple correspond to the expression.

8. δ is a not-null constraint:

   For any vector of terms $\bar{c}$.The formula:

$$\neg \left[N(\bar{c}) \wedge c_{\#_A^N} = null \right]$$

   is true iff:

$$\left[N(\bar{c}) \wedge c_{\#_A^N} = null \right]$$

   is false.

   Suppose that $N(\bar{c})$ is true, then:

   - If $c_{\#_A^N} = null$ is true, a rule of Definition 74 is violated by X, then X is not a model of one of the rules and so we have a conflict.

   We have a conflict when the vector of terms corresponds to the expression, which means that that cannot exist an answer set X where the tuples correspond to the expression.

   Then, we can conclude that $\alpha(X) \models F$ and so $\alpha(X)$ is a repair of D.

□

In order to prove that a repair D' always match with some answer set X, we start by proving that, since the repair is according to the theory, X is according to the program that results from that theory (considering D'), that is, X is a model of that program and also a minimal one.

Theorem 4 (Repair Completeness)**.** *Let $\mathcal{P} = \langle D, D_E, F \rangle$ be a database repair problem. For every repair D' of D, with respect to $\mathcal{P}$, there is an answer set X of $\varphi(\mathcal{P})$ such as $\alpha(X) = M(D')$.*

Proof (sketch). Let $\mathcal{P} = \langle D, D_E, F \rangle$ be a database repair problem, and D' a repair of D, with respect to $\mathcal{P}$, where $D' = D \oplus \Delta$ and an answer set X such that $\alpha(X) = M(D')$.

In order to X be an answer set, with respect to the Definition 23 and then 20, it needs to be a Herbrand model where $X = X'$ and X' is the least model of $ground(P)^X$, such that $\forall_{Y:Y \text{ is a model of } ground(P)} X' \subseteq Y$. Assume that there is a X such that:

$$X = \{\, p_N(t) \mid N(t) \in M(D) \,\} \cup \{\, p\_new_N(t) \mid N(t) \in M(D_E) \,\} \cup$$

$$\{\, p\_update\_single_N^A(t) \mid N(t) \in M(D) \,\} \cup \{\, p\_keep_N(t) \mid N(t) \in M(D') \,\} \cup$$

$$\{\, delete(t_0) \mid N(t) \in M(D) \wedge d(N,t) \in \Delta \,\} \cup$$

$$\{\, n\_delete(t_0) \mid N(t) \in M(D) \wedge d(N,t) \notin \Delta \,\} \cup$$

$$\{\, insert(t_0) \mid N(t) \in M(D_E) \wedge i(N,t) \in \Delta \,\} \cup$$

$$\{\, n\_insert(t_0) \mid N(t) \in M(D_E) \wedge i(N,t) \notin \Delta \,\} \cup$$

$$\{\, p\_update_N^A(t_0), updateAttrVal_N^A(t_0, null) \mid N(t) \in M(D) \wedge u(N,t,A) \in \Delta \,\} \cup$$

$$\{\, n\_update_N^A(t_0), updateAttrVal_N^A(t_0, t_{\#_A^N}) \mid N(t) \in M(D) \wedge u(N,t,A \notin \Delta \,\} \cup$$

$$\{\, update\_aggr_N(t_0,t'), update(t_0) \mid N(t) \in M(D) \wedge t' = u^{aggr}(\Delta,N,t) \wedge \exists_A u(N,t,A) \in \Delta \,\} \cup$$

$$\{\, aux_{N_1,N_2}^{A,B}(t) \mid \exists_{\delta_{fk}=\langle N1,N2,A,B\rangle \wedge \delta_{fk} \in F} \exists_{t'} \forall_{i=1}^{k} aux_{N_1,N_2}^{A_i,B_i}(t,t') \,\} \cup$$

$$\{\, aux_{N_1,N_2}^{A_i,B_i}(t,t') \mid \exists_{\delta_{fk}=\langle N1,N2,A,B\rangle \wedge \delta_{fk} \in F} (t_{\#_{A_i}^{N1}} = null \vee t_{\#_{A_i}^{N1}} = t'_{\#_{B_i}^{N2}}) \,\} \cup$$

$$\{\, aux_N^A(t) \mid \exists_{\delta_{ckc}=\langle N,A,\theta,V\rangle^{ckc} \wedge \delta_{ckc} \in F} (t_{\#_A^N} = null \vee t_{\#_A^N} \theta\ V) \,\} \cup$$

$$\{\, dom_N^A(val) \mid \exists_{\delta_{dc}=\langle N,A,Dom\rangle} val \in Dom \,\}$$

First, we prove that X is a model. In order to be a model, with respect to Definition 20, it needs to entail $\pi(ground(P)^X)$, that is, for every ground rule r of P, when $X \models ground(B(r))$ then $X \models ground(H(r))$. We treat each rule of $ground(P)^X$, determined as in defined in Definition 22, in cases, assuming that $X \models B(r)$ and proving that $X \models H(r)$:

- Rules of Definition 65:

  By definition of X, these facts always belong to it because the only condition is that they are in D or D_E, so they are all satisfied in X.

- Rules of Definition 66:

  With respect to 1., when $delete(t_0) \in X$ then $n\_delete(t_0) \notin X$ and when $n\_delete(t_0) \in X$ then $delete(t_0) \notin X$ because of the cycle made by the rules of the definition, so they are exclusive between them. As, $p_N(t)$ is a fact and always belong to X, then the existent of $delete(t_0)$ or $n\_delete(t_0)$ in X only depends of the cycle. So when $X \models B(r)$ then $X \models H(r)$. About the $p\_keep_N(t)$, it only depends of $delete(t_0) \in X$ or $update(t_0) \in X$, then when $X \models B(r)$ then $X \models H(r)$.

  With respect to 2., when $insert(t_0) \in X$ then $n\_insert(t_0) \notin X$ and when $n\_insert(t_0) \in X$ then $insert(t_0) \notin X$ because of the cycle made by the rules of the definition, so they are exclusive between them. As, $p\_new_N(t)$ is a fact and always belong to X, then the existent of $insert(t_0)$ or $n\_insert(t_0)$ in X only depends of the cycle. So when $X \models B(r)$ then $X \models H(r)$. About the $p\_keep_N(t)$, it only depends of $insert(t_0) \in X$, $delete(t_0) \in X$ or $update(t_0) \in X$, then when $X \models B(r)$ then $X \models H(r)$.

  With respect to 3., when $p\_update_N^A(t_0) \in X$ then $n\_update_N^A(t_0) \notin X$ and when $n\_update_N^A(t_0) \in X$ then $p\_update_N^A(t_0) \notin X$ because of the cycle made by the rules of the definition, so they are exclusive between them. As, $p_N(t)$ is a fact and always belong to X, then the existent of $p\_update_N^A(t_0)$ or $n\_update_N^A(t_0)$ in X only depends of the cycle. So when $X \models B(r)$ then $X \models H(r)$. With respect to 3. c) and d), both predicates cover all the cases, the cases where $p\_update_N^A(t_0) \in X$,

and the cases where $p\_update_N^A(t_0) \notin X$ that is, in all occasions, $X \models H(r)$. With respect to 3. e) and f), they are trivial. About the $p\_keep_N(t)$, it only depends of $update\_aggr_N(t_0, t') \in X$ and $delete(t_0) \in X$ then when $X \models B(r)$ then $X \models H(r)$.

- Rules of Definition 67:

  We assumed that $X \models B(r)$ then $p\_keep_N(t) \in X$ and $p\_keep_N(t') \in X$, which implies that $N(t) \in M(D')$ and $N(t') \in M(D')$ which contradicts Definition 42 and so D' cannot be a repair, which is a contradiction. Then, X would be eliminated and so, $X \models H(r)$.

- Rules of Definition 68:

  We assumed that $X \models B(r)$ then $p\_keep_N(t) \in X$ and $p\_keep_N(t') \in X$, which implies that $N(t) \in M(D')$ and $N(t') \in M(D')$ which contradicts Definition 43 and so D' cannot be a repair, which is a contradiction. Then, X would be eliminated and so, $X \models H(r)$.

- Rules of Definition 69:

  We assumed that $X \models B(r)$ then $p\_keep_N(t) \in X$ and $p\_keep_N(t') \in X$, which implies that $N(t) \in M(D')$ and $N(t') \in M(D')$ which contradicts Definition 44 and so D' cannot be a repair, which is a contradiction. Then, X would be eliminated and so, $X \models H(r)$.

- Rules of Definition 70:

  Here, if $aux_{N_1,N_2}^{A,B}(t) \in X$ then the rule is eliminated. Otherwise, $r' = ground(r)^X = \leftarrow p\_keep_{N1}(t)$. As we assumed that $X \models B(r)$ then $p\_keep_{N1}(t) \in X$, which implies that $N(t) \in M(D')$, but as $aux_{N_1,N_2}^{A,B}(t) \notin X$, it contradicts the Definition 45, and so D' cannot be a repair, which is a contradiction. Then, X would be eliminated and so, $X \models H(r)$.

- Rules of Definition 71:

  Here, if $aux_{N_1,N_2}^{A,B}(t) \in X$ then the rule is eliminated. Otherwise, $r' = ground(r)^X = \leftarrow p\_keep_{N1}(t)$. As we assumed that $X \models B(r)$ then $p\_keep_{N1}(t) \in X$, which implies that $N(t) \in M(D')$, but as $aux_{N_1,N_2}^{A,B}(t) \notin X$, it contradicts the Definition 46, and so D' cannot be a repair, which is a contradiction. Then, X would be eliminated and so, $X \models H(r)$.

- Rules of Definition 72:

  Here, if $aux_N^A(t) \in X$ then the rule is eliminated. Otherwise, $r' = ground(r)^X = \leftarrow p\_keep_N(t)$. As we assumed that $X \models B(r)$ then $p\_keep_N(t) \in X$, which implies that $N(t) \in M(D')$, but as $aux_N^A(t) \notin X$, it contradicts the Definition 47, and so D' cannot be a repair, which is a contradiction. Then, X would be eliminated and so, $X \models H(r)$.

- Rules of Definition 73:

  Here, if $dom_N^A(val) \in X$ then the rule is eliminated. Otherwise, $r' = ground(r)^X = \leftarrow p\_keep_N(t) \wedge t_{\#_A^N} \neq null$. As we assumed that $X \models B(r)$ then $p\_keep_N(t) \in X$ and $t_{\#_A^N} \neq null$, which implies that $N(t) \in M(D')$, but as $dom_N^A(val) \notin X$, it contradicts the Definition 48, and so D' cannot be a repair, which is a contradiction. Then, X would be eliminated and so, $X \models H(r)$.

- Rules of Definition 74:

  $r' = ground(r)^X = r$. We assumed that $X \models B(r)$ then $p\_keep_N(t) \in X$ and $x_{\#_A^N} = null \in X$, which implies that $N(t) \in M(D')$ which contradicts Definition 49 and so D' cannot be a repair, which is a contradiction. Then, X would be eliminated and so, $X \models H(r)$.

Then, for every considered rule, X follows that when $X \models B(r)$ also $X \models H(r)$, and so X is a model. Now, we check if X coincides with the least model. If it is a least model, then $X \subseteq I$, for all models I. We separate every predicate that may belong to X by cases:

- $p_N(t) \in X$, $p\_new_N(t) \in X$, $p\_update\_single_N^A(t_0) \in X$

  If $p_N(t) \in X$ then $p_N(t) \in I$ because $p_N(t)$ is a fact and otherwise I would not be a model. Same if $p\_new_N(t) \in X$ and $p\_update\_single_N^A(t_0) \in X$.

- $delete(t_0) \in X$

  If $delete(t_0) \in X$ then $n\_delete(t_0) \notin X$ and $update(t_0) \notin X$, then the reduct of the corresponding rule is $delete(t_0) \leftarrow p_N(t)$. As $p_N(t) \in I$, because it is a model of the same program regarding X, then $delete(t_0) \in I$.

- $n\_delete(t_0) \in X$

  If $n\_delete(t_0) \in X$ then $delete(t_0) \notin X$, then the reduct of the corresponding rule is $n\_delete(t_0) \leftarrow p_N(t)$. As $p_N(t) \in I$, because it is a model of the same program regarding X then $n\_delete(t_0) \in I$.

- $insert(t_0) \in X$

  If $insert(t_0) \in X$ then $n\_insert(t_0) \notin X$ then the reduct of the corresponding rule is $insert(t_0) \leftarrow p\_new_N(t)$. As $p\_new_N(t) \in I$, because it is a model of the same program regarding X, then $insert(t_0) \in I$.

- $n\_insert(t_0) \in X$

  If $n\_insert(t_0) \in X$ then $insert(t_0) \notin X$, then the reduct of the corresponding rule is $n\_insert(t_0) \leftarrow p\_new_N(t)$. As $p\_new_N(t) \in I$, because it is a model of the same program regarding X, then $n\_insert(t_0) \in I$.

- $p\_update_N^A(t_0) \in X$

  If $p\_update_N^A(t_0) \in X$ then $n\_update_N^A(t_0) \notin X$ then the reduct of the corresponding rule is $p\_update_N^A(t_0) \leftarrow p_N(t) \wedge p\_update\_single_N^A(t_0)$. As $p_N(t) \in I$ and $p\_update\_single_N^A(t_0) \in I$, because it is a model of the same program regarding X, then $p\_update_N^A \in I$.

- $n\_update_N^A(t_0) \in X$

  If $n\_update_N^A(t_0) \in X$ then $p\_update_N^A(t_0) \notin X$ then the reduct of the corresponding rule is $n\_update_N^A(t_0) \leftarrow p_N(t) \wedge p\_update\_single_N^A(t_0)$. As $p_N(t) \in I$ and $p\_update\_single_N^A(t_0) \in I$, because it is a model of the same program regarding X, then $n\_update_N^A \in I$.

- $updateAttrVal_N^A(t_0, t_{\#_A^N}) \in X$

  If $updateAttrVal_N^A(t_0, t_{\#_A^N}) \in X$ then $p\_update_N^A(t_0) \notin X$ then the reduct of the corresponding rule is $updateAttrVal_N^A(t_0, t_{\#_A^N}) \leftarrow p_N(t)$. As $p_N(t) \in I$, because it is a model of the same program regarding X, then $updateAttrVal_N^A(t_0, t_{\#_A^N}) \in I$.

- $updateAttrVal_N^A(t_0, null) \in X$

  If $updateAttrVal_N^A(t_0, null) \in X$ then the reduct of the corresponding rule is

  $updateAttrVal_N^A(t_0, null) \leftarrow p_N(t) \wedge p\_update_N^A(t_0)$. As $p_N(t) \in I$, because it is a model of the same program regarding X, and all $p\_update_N^A(t_0) \in X$ also $p\_update_N^A(t_0) \in I$ then $updateAttrVal_N^A(t_0, null) \in I$.

- $update\_aggr_N(t_0, t') \in X$

  If $update\_aggr_N(t_0, t') \in X$, since all the predicates of the reduct of the rule that belong to X also belong to I then, $update\_aggr_N(t_0, t') \in I$.

- $update(t_0) \in X$

  If $update(t_0) \in X$, since all the predicates of the reduct of the rule that belong to X also belong to I then, $update(t_0) \in I$.

- $p\_keep_N(t) \in X$

  If $p\_keep_N(t) \in X$, since all the predicates of the reduct of the rule that belong to X also belong to I then, $p\_keep_N(t) \in I$.

- $aux_{N_1,N_2}^{A,B}(t) \in X$

  Since all $p\_keep_N(t) \in X$ also $p\_keep_N(t) \in I$, then all $aux_{N_1,N_2}^{A,B}(t) \in X$ also $aux_{N_1,N_2}^{A,B}(t) \in I$.

- $aux_{N_1,N_2}^{A_i,B_i}(t, t') \in X$

  Since all $p\_keep_N(t) \in X$ also $p\_keep_N(t) \in I$, then all $aux_{N_1,N_2}^{A_i,B_i}(t, t') \in X$ also $aux_{N_1,N_2}^{A_i,B_i}(t, t') \in I$.

- $aux_N^A(t) \in X'$

  Since all $p\_keep_N(t) \in X$ also $p\_keep_N(t) \in I$, then all $aux_N^A(t) \in X$ also $aux_N^A(t) \in I$.

- $dom_N^A(val) \in X'$

  Since both X and I are models of the same program, they consider the same set of ICs, F, and so all δ_{dc} consider by X are also consider by I, and so, all $dom_N^A(val) \in X$ also $dom_N^A(val) \in I$.

Then, for any model I, $X \subseteq I$, coinciding with the least model.

At last, it is needed to prove that $\alpha(X) = M(D')$. That corresponds to $\alpha(X) \subset M(D')$ and $M(D') \subset \alpha(X)$. That $M(D') \subset \alpha(X)$ is easy to check to be true because all $N(t)$ such that $N(t) \in M(D')$ were added to X with $p\_keep_N(t)$ atoms. Also, we do not added more tuples to X so $\alpha(X) = M(D')$. □

Theorem 5 (Repair Soundness and Completeness)**.** *Let $\mathcal{P} = \langle D, D_E, F \rangle$ be a database repair problem. If and only if X is an answer set of $\varphi(\mathcal{P})$, then $\alpha(X)$ is a repair of D with respect to $\mathcal{P}$.*

Proof (sketch). Let $\mathcal{P} = \langle D, D_E, F \rangle$ be a database repair problem. There is an answer set X regarding $\varphi(\mathcal{P})$, such that $\alpha(X)$ is a repair of D with respect to $\mathcal{P}$. This follows from Theorem 3 and from Theorem 4. □

5.2 Determining the minimal repairs

Since we allow the user to choose a minimality criteria, we have to filter the answer sets previous determined in order to show to the user only the minimal ones with respect to the chosen criteria. To accomplish that, since the implementation becomes difficult and increases the computational complexity, we use a collection of encodings named Metasp. In order to use it we only need to make use of the provided encodings, change the calls in the command line (that is used to execute the program) and add a line with the *minimize* predicate to the program file. The *minimize* predicate has the following form [Bar01]:

$$\#minimize[l_1 = w_1@J_1, ..., l_k = w_k@J_k]$$

where every l_i is a literal, every w_i is an integer weight and J_i is an integer priority level. However, with respect to cardinality and under-set inclusion minimal criteria we only use *minimize* as:

$$\#minimize[l_1, ..., l_k]$$

After we have the program and execute it, the obtained results do not assume the form that we are expecting. They are encapsulated in other predicates and we have to extract those results.

Definition 77 (Extracting Function from Metasp Results)**.** Let $\mathcal{P} = \langle D, D_E, F \rangle$ be a database repair problem and let X be an answer set of $\varphi(\mathcal{P})$ executed with the Metasp encodings. The extracting function α^m is defined as:

$$\alpha^m_N(X) = \{p\_keep_N(\bar{c}) \mid hold(atom(p\_keep_N(\bar{c}))) \in X\}$$

$$\alpha^m(X) = \bigcup_{N \in \mathcal{N}_D} \alpha^m_N(X)$$

The function $\Delta_m(X)$, that gets the modifications with respect to the original database considering X, is defined as:

$$\Delta_m(X) = \{\, d(N,t) \mid hold(atom(delete(t_0))) \in X \wedge hold(atom(p_N(t))) \in X \,\}$$

$$\cup \{\, i(N,t) \mid hold(atom(insert(t_0))) \in X \wedge hold(atom(p\_new_N(t))) \in X \,\}$$

$$\cup \{\, u(N,t,A) \mid hold(atom(p\_update^A_N(t_0))) \in X \wedge hold(atom(p_N(t))) \in X \,\}$$

However, this tool only helps with the cardinality and under-set inclusion minimality where the weight of the considered operations to minimize has to be equal to all types, leaving us with the weight minimality to deal with, where the weight can vary according to each operation type. In order to deal with this criteria, we start by applying a different use of the *minimize* predicate with the following form:

$$\#minimize[l_1 = w_1, ..., l_k = w_k]$$

where each l_i represent a different operation type and w_i the corresponding weight value. Then, when we execute the program, it gives us the answers, each one associated with an optimization value that is the sum of the weights of all operations contained in each answer set. However, only the last obtained answer has an optimal weight as opposed to the other. So, in order to get all other answers with the optimal value, we use the following parameter in the command line call to get only answers with the optimal sum of weights value:

opt-all=optimalValue

This allows to just retrieve the answers which optimization value is the optimal one.

Here, predicates are not encapsulated with other predicates and so the extracting function is the following:

Definition 78 (Extracting Function with Cardinality and Weight Minimality)**.** Let $\mathcal{P} = \langle D, D_E, F \rangle$ be a database repair problem and let X be an answer set of $\varphi(\mathcal{P})$ obtained by using weight minimality criteria. The extracting function α^{wm} is defined as

$$\alpha_N^{wm}(X) = \{p\_keep_N(\overline{c}) \mid p\_keep_N(\overline{c}) \in X\}$$

$$\alpha^{wm}(X) = \bigcup_{N \in \mathcal{N}_D} \alpha_N^{wm}(X)$$

The function $\Delta_{wm}(X)$, that gets the modifications with respect to the original database considering X, is defined as:

$$\Delta(X) = \{\, d(N,t) \mid delete(t_0) \in X \wedge p_N(t) \in X \,\}$$

$$\cup\ \{\, i(N,t) \mid insert(t_0) \in X \wedge p\_new_N(t) \in X \,\}$$

$$\cup\ \{\, u(N,t,A) \mid p\_update_N^A(t_0) \in X \wedge p_N(t) \in X \,\}$$

With the last two definitions, we know how to extract the answers of the program regardless the minimality criteria chosen. We only need to extract the relevant predicates. With those, we also know the operations made (insert, delete and update predicates) and which are the tuples that remain in the database after the repairing process (p_keep predicate).

6

The DRSys application

We already defined the ASP code that helps to get a database's repairs so now we describe the developed application that supports the interaction with the user and generates the ASP program regarding the repair problem. This application allows to connect to a PostgreSQL database, retrieve its data, specify new ICs and store the chosen repair in the database, changing it. In order to achieve that, it presents a sequence of screens where the user can specify the ICs semantics, view and edit the generated ASP code, provide information to generate new tuples to be inserted, limit the delete and update operations, choose the minimality criteria, and more.

This chapter is dedicated to the application[1], namely, its architecture in order to understand how each component interact with others, the provided features, the technical decisions and also some examples to show how the application matches the expected results for some ICs depending on the chosen semantics.

6.1 Architecture

The application is organized in components. They interact with each other but the application does not accomplish its goals without interacting with external entities. In order to illustrate the entities that interact with the system, here is a schematic diagram:

[1] Application available upon request: lr.luis@campus.fct.unl.pt

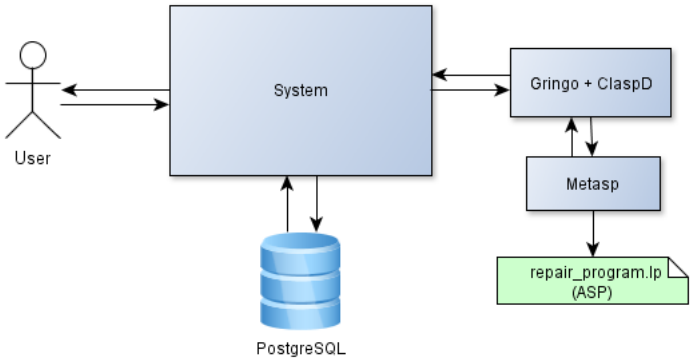

Figure 6.1: Architecture - Interaction with external entities

The system has to interact with the user so that she can insert new ICs and make the decisions associated with the insert, delete and update operations, minimality criteria and so on. Also, it has to interact with the database specified by the user in order to retrieve data and already stored ICs. Finally, in order to determine the repairs, it generates a file with the ASP program, uses a grounder (Gringo), a compiler (clasp) and a tool to retrieve minimal repairs (Metasp). Gringo is a grounder that translates logic programs into equivalent propositional logic programs. The answer sets of such programs can be computed by a solver, e.g. clasp [GS]. Metasp is used to choose the answers sets that are minimal and it is executed when the application runs the ASP program, piping the output of Gringo back into Gringo and then running it in conjunction with meta-encodings.

The system itself is also divided in components in order to separate the different type of tasks namely the input processing, conversion of the ICs to ASP code, check if the database is consistent with the ICs, execute and process the ASP program answers. It has the following detailed architecture:

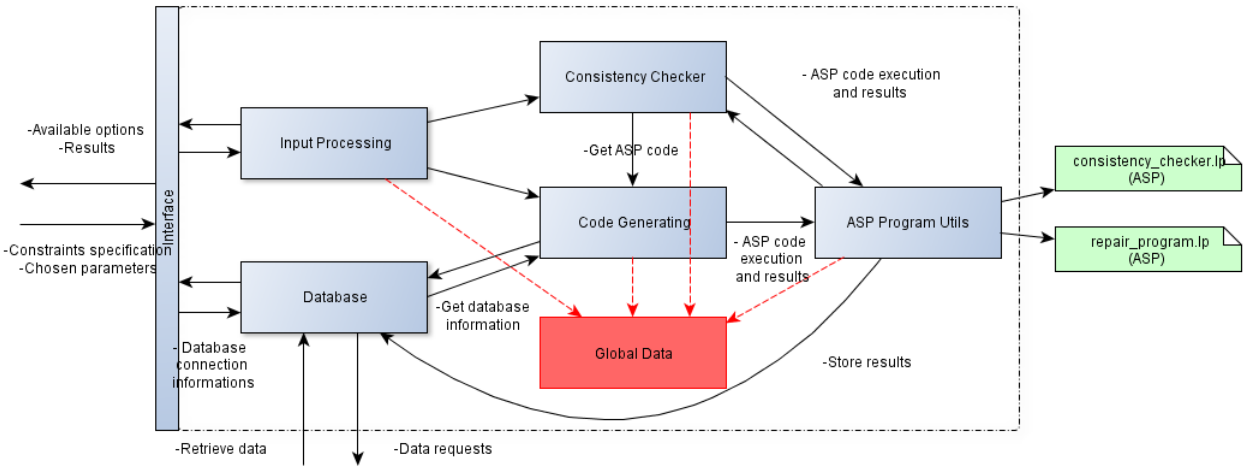

Figure 6.2: Architecture - Components

The responsible for beginning the execution of the system is the user which interacts with the interface that is implemented in Java (and also partitioned in components that represent the different screens) and then the latter starts the interaction with the components of the system. First, the user interacts with the database by specifying the needed information to do the connection, but not directly. He interacts with a component that represents the database, the **Database** one, that offers the operations necessary to do the connection and retrieve information from the database namely the already stored ICs. Then, everything that is specified by the user in the interface is processed by the **Input Processing** module that is responsible for parsing and processing all the information inserted in the interface (e.g. the new ICs) and interacts with the other modules for the execution of some procedures that make use of the parsed information. Namely, it processes the information about the new tuples that can be inserted, so that the **Constraint Checker** and **Code Generating** modules can use them in the ASP code. The **Constraint Checker** module creates an ASP program that checks if the database content respects the ICs defined by the user in the interface. To build the program, it uses the **Code Generating** module that generates all the ASP code for the tuples of the database, the generating rules and also the ICs code. The **Code Generating** module also interacts with the **Database** module in order to obtain the ICs that were defined in the DBMS. Both **Constraint Checker** and **Code Generating** modules interact with another module that contains some methods to help with some string processing needs. The **ASP Program Utils** module is responsible for getting the ASP code and write it in the corresponding file, also executes the program and extract the answers when it is requested by the **Code Generating** module. These answers are also retrieved by the **Constraint Checker** in order to conclude if the database is already consistent with the ICs or not. The **ASP Program Utils** module also stores the ICs in the database, in the "constraints" table, and uses the *p_deleted*, *p_inserted*, *p_was_updated* and *p_updated* predicates to obtain the operations that need to be performed in corresponding tables in order to retrieve the consistency. Finally, there is a module called **Global Data** that is responsible for keeping the data structures that are common to all other modules.

6.2 Features

Now that we know the composition of the system and how its components interact, we proceed to the explanation of the features and how to interact with the interface in order to make use of them.

The first step consists on specifying the necessary information to connect to the database. It is inserted in the first screen of the application, the connection screen, where the user is asked to insert the host name, the database name, user name and password with respect to an existing database created on the DBMS. Also, the user can specify if she is a basic user or an expert user. In the case of being a basic user, she only sees the screen to specify the ICs, the corresponding semantics and the allowed operations (insert, delete or update

operations) and the screen where the user can choose the minimality criteria, etc.

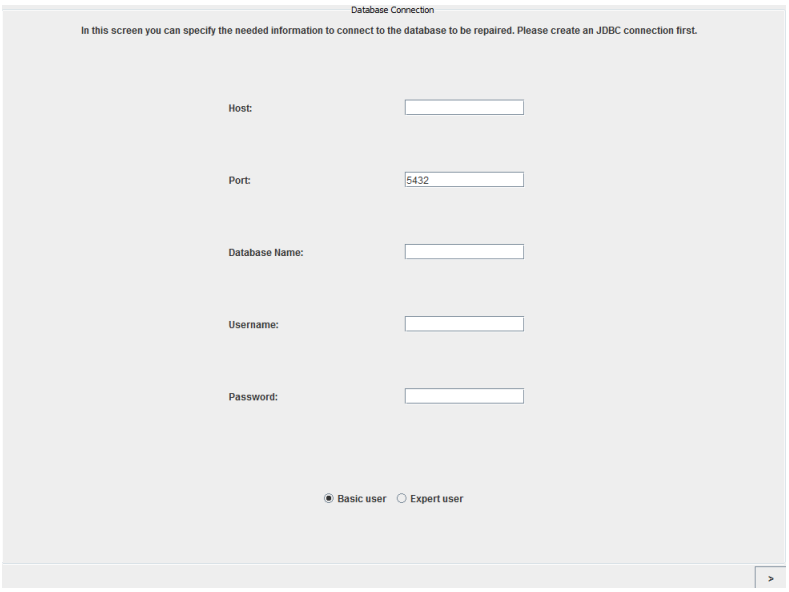

Figure 6.3: Connection screen

6.2.1 ICs specification screen

After the connection to the database, there is a screen that is common to both user types: the constraints specification screen. In this screen we present the options that allow to specify the different IC types. The user can specify the type of the IC (e.g. primary key) and then she can enter the involved attributes by clicking in their names in the left panel and filling the corresponding boxes. About the check constraints, in this document, they are defined as simple constraints that only involve one attribute, one operator and one value however, in the application, we allow to define a little more complex ones that may involve more than one attribute of the same table, multiple operators and multiple values.

Besides, in this screen, we also allow the user to choose the semantics to be applied to each specified IC. We chose to give that option to the user based on the analysis made on Chapter 4. In that chapter, we stated that some alternatives were better than others but we also observed some examples where we wanted to use a semantics that usually is not the more desired one. So, in order to provide good results to the user for whatever the rules to be modeled, we provide the not so usual semantics so that the user can choose what is better for each situation. In the case of the user does not know technical details, then by not changing the default definitions, the recommended ones that were specified in Chapter 4 are applied.

As mentioned before, in this screen, it is also possible to choose the allowed operations.

After all these choices, the user can add the new IC to the list of defined ICs and then

proceed for the next screen or just add more ICs.

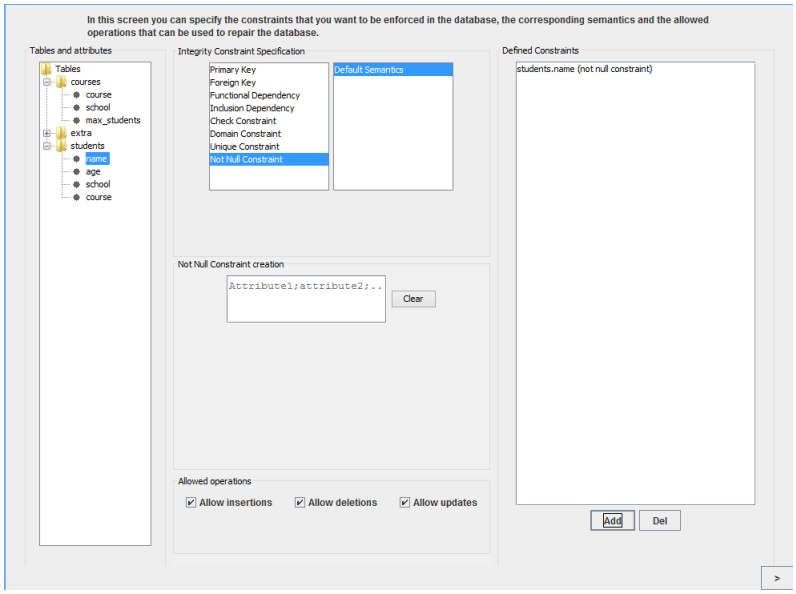

Figure 6.4: Constraints screen

The next screen is a trivial one, to an expert user, where she can see the generated code (with the new ICs) so that she can have an overall notion of the program to keep in mind when doing the choices that come next.

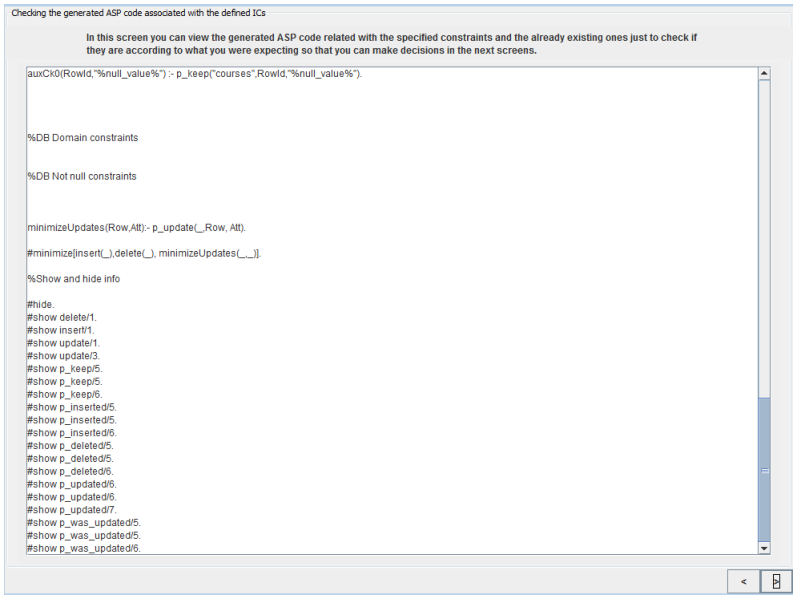

Figure 6.5: View ASP screen

The next screen allows to insert some additional ASP code to merge with the generated one. This gives some freedom to the user by allowing to specify other criteria that are not provided by the application like to treat some specific case with a constraint, for example, or just add another constraint type different from the ones provided in the previous screen, as we show in the next example.

Example 16. Considering the database:

ID	*Name*	*Location*	*AvailableChairs*
C1	*Blue Cinema*	*null*	*60*
C3	*Red Cinema*	*Lisbon*	*50*

Table 6.1: Cinema table

ID	*Name*	*Location*	*Director*	*AvailableChairs*
T1	*Green Theater*	*Porto*	*null*	*20*

Table 6.2: Theater table

For this example, we could add some new constraint like:

$$\leftarrow p\_keep_{Cinema}(CID, CName, CLocation, CAval),$$

$$p\_keep_{Theater}(TID, TName, TLocation, TDirector, TAval), TAval > CAval.$$

This constraint expresses that we cannot have theaters with more available places than some cinema, that is, it is a check constraint that refers to two tables.

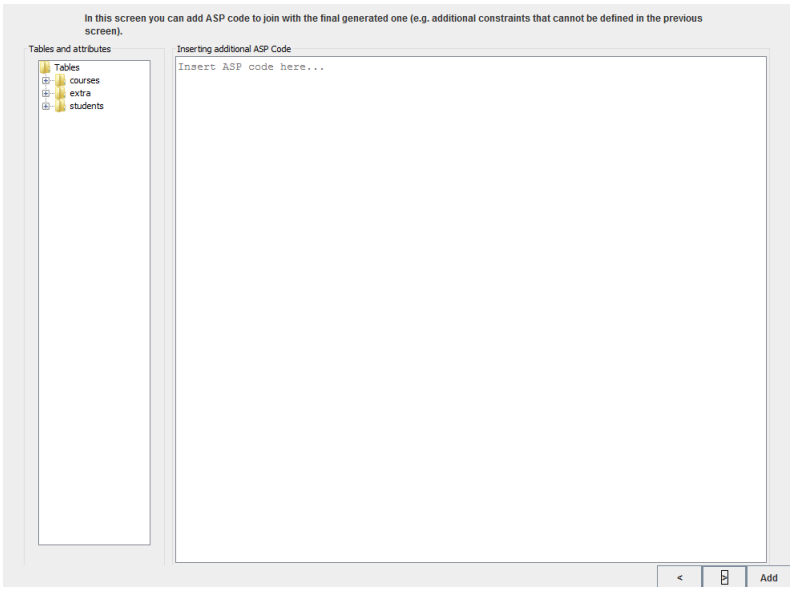

Figure 6.6: Add ASP screen

6.2.2 Insert options screen

In the insertion process we have to know what are the combinations of values that form the tuples that can be inserted in the database in order to repair it. With the purpose of getting all the tuples that can be inserted avoiding the effort (to the user) to provide

another supplemental database or to specify all the possible tuples by hand, we present three different ways of specifying new tuples.

The first one is the simplest. It allows to add tuples by hand or select them from another table of the same database.

The second one allows the user to specify a list of values to assign to an attribute for insertion purposes. Those values are the ones that can be used for that attribute in new tuples and each of these tuples is a possible combination of values of the attributes that compose them. In the insertions screen, the attribute can be specified in the left box and the allowed values for that attribute in the right box.

The third one allows the user to specify that an attribute (specified in the left box) can have all the values of the attribute specified in the right box. These values are the ones that are present in the database tuples for that specific attribute. This option allows to specify that an attribute can have exactly the same values as other attribute, as an inclusion dependency or an foreign key.

These last two options allow to generate tuples with all the combinations of values that were specified forming the tuples that are possible to be inserted.

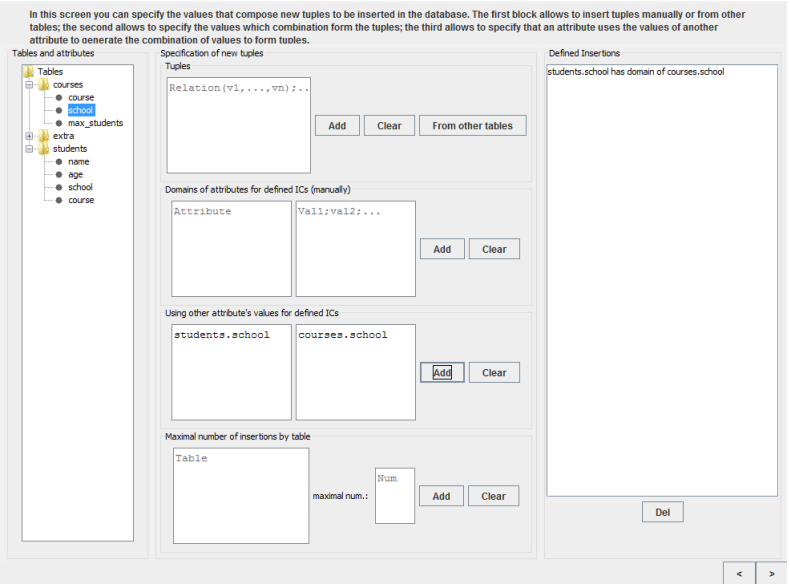

Figure 6.7: Insertion operations screen

Also in this screen, it is possible to specify the maximal number of insert operations that are allowed to do in a specific table. This is useful when the user prefers not to do a big number of insertions, because she wants to have a balanced number of operations or just because she does not want to increase much the size of the table. If that limit is not specified for a table, it means that we can do any number of insert operations in that table. If the user specify more than one limit to the same table, the latter is the one that is used.

6.2.3 Delete options screen

After the options regarding insert operations, we have the options related to delete operations. This screen is divided in three sections.

The first allows to specify the maximal number of delete operations of a specific table as in insert options screen, that is useful when the user does not want to lose a lot of data of a specific table, for example.

The second allows to specify tables from where we cannot delete any tuple. For example, we could have a database that has information about stores and its costumers, workers, suppliers, products, then we would not want to simply remove existent stores from stores table if they still exist.

The third option allows to view the database tuples separated by table and select tuples from different tables that we do not allow to be deleted, for example, when some specific tuple has information that must remain. Returning to the previous example, we could not want, for example, to remove some specific store but we do not mind if others are removed because they no longer exist.

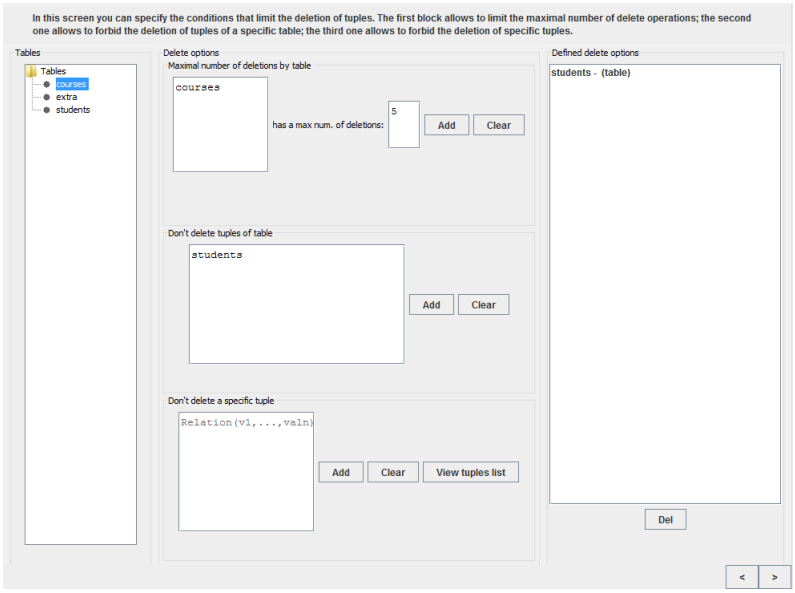

Figure 6.8: Delete operations screen

After this screen, we also have one that has the options related to the update operations. These options are the same that we have for the deletion operations but applied to the update ones. The reasons are the same as before but in this case we may do not want to introduce more null values into the database and so we limit the number of update operations.

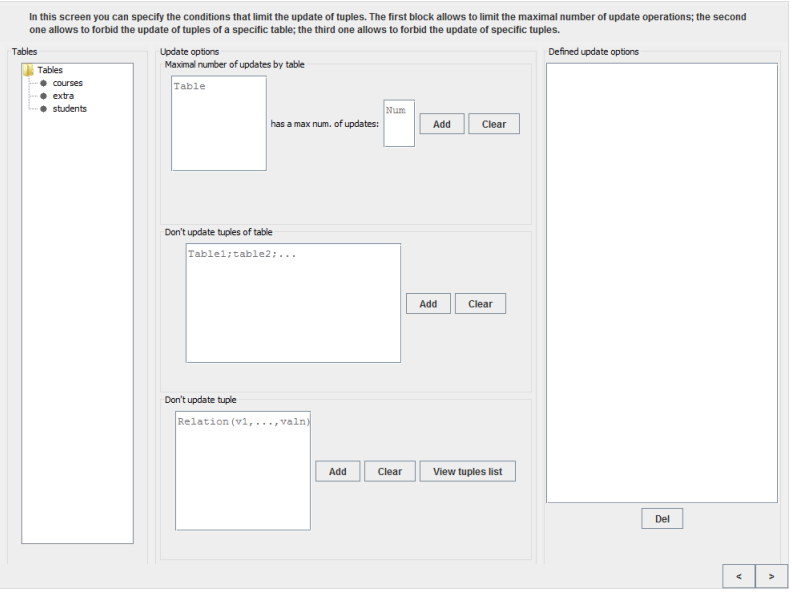

Figure 6.9: Update operations screen

Besides the options related to operations, there is also a screen where the user can choose another options that are independent of the type of the operations.

6.2.4 Other options screen

Here, we present more options to constraint the set of repairs that appear in a following screen.

First, the user can specify the chosen minimality criteria and she has to choose exactly one. It is possible to choose under set, cardinality or weight minimality. When the weight one is chosen, then the user can choose to use the default weight values or assign new ones.

We also let the user choose if she wants to see only the repairs that are obtained by updates or insertions where the irrelevant attributes values (the ones that not affect the result) are null values, or insertions where the attributes are non-null values.

Besides these two options, we also allow the user to specify an overall maximal number of operations (the sum of delete, insert and update operations) that cannot be exceeded by any repair. Also, it is possible to specify the maximal number of operations for a specific table in the case of the user wanting to keep a specific table more "stable" than others, with less information changing.

Finally, there is an option that allows the user to choose if she allows to save the defined ICs in a "constraints" table or not. Allowing it saves work for next executions of the application where the previous defined ICs are uploaded to the current ASP program. However, the user may not want to change the database schema by adding an extra unrelated table.

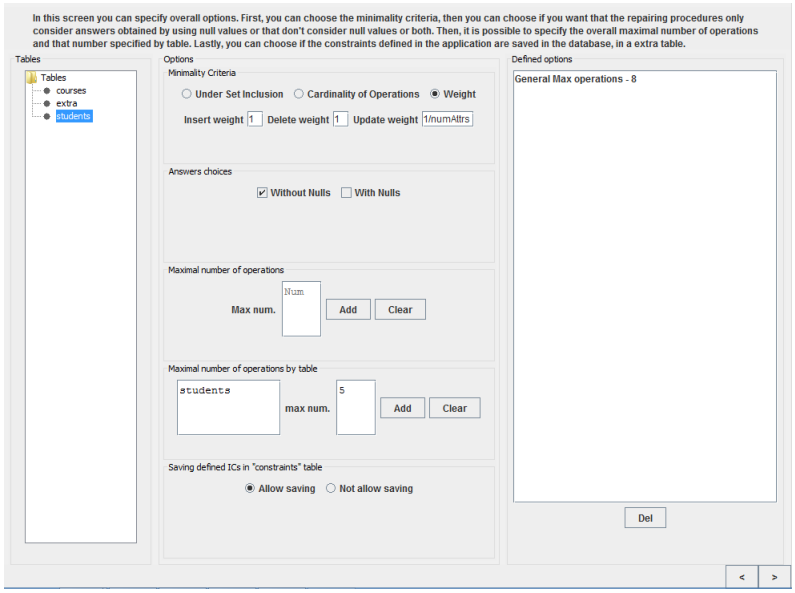

Figure 6.10: Other options screen

The next screen is about editing the final generated ASP code because the user may want to change or remove code (e.g. some constraint). Here, we show the code that corresponds to the tables that are involved in the ICs, the corresponding tuples, the new ICs, database's ICs code and the code that results from the user's options.

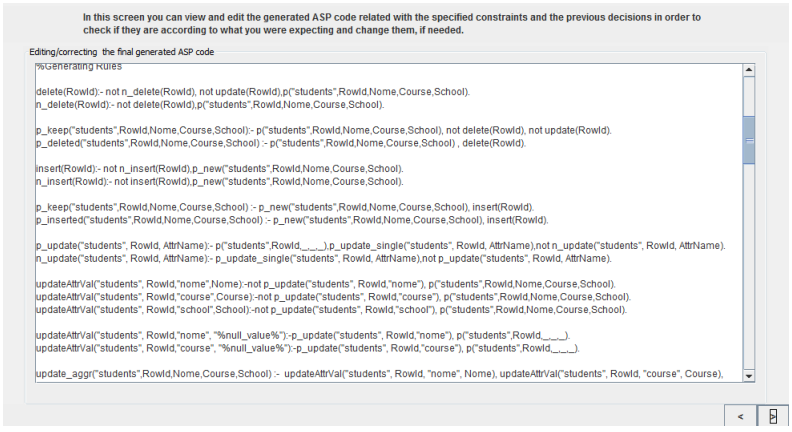

Figure 6.11: Edit ASP code screen

If the database is already consistent with the defined ICs, then instead of showing the generated code, we present a message stating that the database is already consistent.

The last step is about executing the final code and the repairs are obtained. So, the resulting repairs are presented to the user and she can choose to expand them in order to see which are the operations involved in each possible repair and then choose one to be performed in the database.

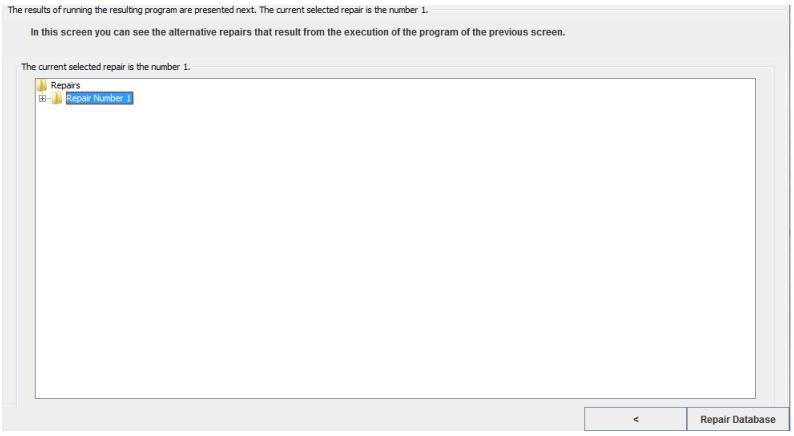

Figure 6.12: Results screen

6.3 Decisions

In order to implement the presented application, we had to choose the adequate tools considering the technical needs. Namely, about the implementation of the interface, about the implementation in ASP and about the used DBMS. Also, after the technical decisions, we had to make some decisions in order to determine the needs of the user that we want to satisfy and so provide the adequate features.

6.3.1 Tools

First, about the interface, it is implemented in Java since it provides an easy way to develop interfaces. We also make use of xswingx[2] to help in some interface details. Also, Java supports connection to some DBMSs in order to obtain information about the database and the corresponding data. By using Java, we can connect both the DBMSs and the grounder+solver since we can use Java to connect to the database, to generate the ASP code and execute it by using the command line, the grounder and the solver.

After the programming languages decisions, we determine the DBMS in which the application is based on because all DBMSs have different syntaxes (that vary when we want to do insertion, deletion or update operations) and different features that sometimes are related to ICs. Here, we only analyze the more popular DBMSs.

Starting with PostgreSQL and MySQL (only InnoDB engine), regarding the supported constraints, the former and the latter support the definition of primary keys, foreign keys, unique constraints and not-null constraints. Besides both support the definition of check constraints, only PostgreSQL actually verify the data to check if the constraint is being ensured and implement domain constraints. With respect to optimization techniques each one has its own and it is not relevant to analyze them. About Oracle, it is similar with PostgreSQL but the latter is free and it has Java support.

[2]https://code.google.com/p/xswingx

About the grounder and the solver, we use Gringo and clasp (and claspD) respectively. This decision is due to our minimality criteria needs. About Gringo and claspD, they are the tools that are supported by Metasp. Metasp is a collection of encodings that are responsible for retrieving the minimal repairs and it is ready just for Gringo and claspD. Metasp helps in avoiding the difficulty of implementing an ASP program responsible by the optimization part. It allows to determine the under-set inclusion and cardinality minimal repairs, but it is not the only way of getting answers regarding cardinality criteria, it is also possible to get them by using just the *minimize* predicate (as in weight minimality). Considering these two options with respect to cardinality criteria, we decided just to use Metasp with under set inclusion criteria because, after experimenting both options, we concluded that we can get cardinality minimal answers faster only with the *minimize* predicate than with Metasp, as it is used in [Alv11].

About clasp, it supports the *opt-all* parameter that is used in cardinality and weight minimality. We need this parameter because with the *minimize* predicate, we first get some answers that are not minimal and as we wait to get more answers from the solver, they are better (more accurate) than the previous ones and so, only the last obtained answer is the minimal that we were waiting for. By using the parameter, after we know one of the optimal (minimal) answers (and its corresponding optimization value), we run the program again but this time with the parameter using the optimization value, and we get all the answers that have that optimal value (that corresponds to the sum of weights of the operations that compose that repair).

6.3.2 Main features

There are also some other decisions that are related more specifically with the provided features of the application.

In the first screen, the user is asked to specify if she is a basic or an expert user. It was decided to provide this option because the application has a lot of screens where, for an inexperienced user, it may be confusing or unnecessary and so, in the basic version, only the essential screens are shown.

About the insert operations, it was decided to generate new tuples that were specified manually, based on a specified list of values and domains of other attributes. If the allowed values are few and the user knows the combination of values that form the allowed tuples, then specifying them by hand could be a good option. Also, it is possible to use the tuples of an extra table (e.g. that contains data that is no longer used but that stills works for insertion purposes).

However, there are cases where the user does not know the specific combination of values that form the tuples but she knows the allowed values for each attribute and so, we chose to provide an option where the user can specify the list of allowed values for each attribute and then the program does the combination of all these values and the resulting combinations form the new tuples. This is useful when specifying each tuple

one by one is very exhausting for the user or the user just do not know exactly the possible combination of attribute values and so, specifying the allowed values for each attribute, it is faster for the user.

There are other cases, where we want that an attribute assumes all the values of other attribute and in that case, just referring that both have the same domain, is much more faster than specifying each value by hand. This is useful for cases where specifying foreign keys or inclusion dependencies does not fit because the referred values can be moved from time to time and so, when that happens, we would have to delete the tuples that were referring them.

In the last two options, the combination of values is made in a different ASP program file from where the resulting tuples are extracted and then joined with the main repairing code in the form of *p_new* predicates. However, it may seem that we just calculate all the possible combination of values of the different attributes but it is not the case. We just generate a predicate when it is useful to repair some IC. This code is only generated with respect to foreign keys and inclusion dependencies since they are the only constraints where the insertion of new tuples can repair an inconsistency. This is also what is made with the updates.

We generate all the possible updates only for the relevant attributes. In order to determine what are the relevant attributes, the update code is processed each time the user returns to the ICs definition screen and if a specific attribute appears in some IC, then it is a relevant attribute.

Moving on to one of the final steps, there is a screen where the user can see multiple general options, for example, if she wants only to see answers with null values or not-null values. Here is an example where we prefer to only get the repairs with null values:

Example 17. Considering the inclusion dependency IC

$$\langle Employee, Salary, \langle Name\rangle, \langle Name\rangle\rangle^{incd}$$

and the database:

Name	*Address*	*Birth_Date*
John	*Streetnumber1*	$05-04-1990$
Claire	*Streetnumber2*	$07-06-1990$

Table 6.3: Employees table

Name	*Value*
John	*600*

Table 6.4: Salaries table

Here, the first repair consists on removing the second tuple from the $Employees$ table and the second repair consists on inserting a new tuple in $Salaries$ table. We know that the tuple to be inserted has to have the value " $Claire$ " for the $Name$ attribute in order to respect the IC. But what about the $Value$ attribute? We can assume the value 300, 400, 500, 650... There are a lot of possible values that make the IC consistent and we do not know which. If the user specify a range of possible values, from 200 to 600 for example, then we get a lot of possible repairs and the computation would need a lot of time. In this case, we prefer a repair where the $Value$ attribute has a null value instead of getting a set of repairs with different irrelevant values, that would increase the computation complexity.

However, this not always happens:

Example 18. Considering the inclusion dependency IC

$$\langle Order, Product, \langle ProductID \rangle, \langle ProductID \rangle \rangle^{incd}$$

and the following database:

ID	$ProductID$	$Date$	$Address$
$O1$	$P1$	$04-05-2013$	$Street1$
$O2$	$P2$	$06-07-2013$	$Street2$

Table 6.5: Orders table

$ProductID$	$isAvailable$
$P1$	yes

Table 6.6: Products table

In this case, the first repair consists on removing the second tuple from $Orders$ table and the second repair consists on inserting a new tuple into $Products$ table. We already know that for the new tuple, the $ProductID$ attribute has to have the value " $P2$ " but we do not know the value for the $isAvailable$ attribute. However, this attribute only allows the " yes " or " no " values, so the alternative repairs only have two distinct values for this attribute and it does not increase the computation complexity. So, we can ask for the repairs with the non-null values and not the null ones.

Also in the same screen, it is possible to chose the weight minimal criteria. We chose to provide this option to allow the preference of operations over others, giving a lower weight values for priority operations and so we get repairs where the preferred operation type appears more times than others.

There are assigned default values to the operations. In these default values we consider that an insert and a delete operation have 1 as weight value. However, the update

operations are consider to have a weight of $\frac{1}{numOfAttributes}$ value, which means that the weight value of each update operation depends on the table that is being updated, that is, if we are updating a table with four attributes, then an update operation would have a weigh of $\frac{1}{4}$. But why these have so specific weight values? Why they have a default value lower than the other operations? In the case of deleting a tuple from a table with four attributes, we are changing all the values of the tuple (deleting them), that is, we are changing the four attributes. In the case of updating only a value to null (since we consider that each update only changes on attribute to null), then we are only changing one value of four values (to null), that is $\frac{1}{4}$. An update only has the same weight as a delete one when we update the four attributes (as in delete), that is, we have four updates to the same tuple ($\frac{1}{4} + \frac{1}{4} + \frac{1}{4} + \frac{1}{4} = 1$) and in that case, the update operation has the same weight as a delete operation, because they change the same quantity of data.

However, we cannot apply these values directly. Clasp does not allow to use non-integer weight values and Gringo just rounds the weights to integer numbers which may cause to getting repairs that are not update-minimal. In order to solve that, we multiply a single number by each weight value that is specified in *minimize* statement, in order to convert the non-integer numbers to numbers bigger than 0 and then round them, and so, Gringo can deal with them because they are already integer numbers that are higher than 0. That single number to multiply is the Least Common Multiplier (LCM) of the number of attributes of each table that is included in the logic program (because the number of attributes of each table determines the corresponding weight value).

Now about storing results, in order to save them in the database, we use some additional predicates that were already mentioned before: the *p_inserted* predicate is for knowing the values of each inserted tuple; the *p_deleted* predicate is for knowing the values of each deleted tuple; the *p_was_updated* predicate and the *p_updated* predicate are for knowing the values of each tuple that was update and should be updated in the database and we also need to know those new values. Directly extracting these predicates from the chosen repair, we get all the information that is needed to change the database avoiding the access to database information.

About saving the new ICs, we chose to store the new ICs defined by the user in a table. This is useful in order to the user know at any time what were the ICs previously defined and also for future repairing procedures. This information about ICs is not stored in system tables since the user may not have permissions. Also, these ICs are stored as the ones specified directly in the DBMS (in a different and simpler table but with the same essential attributes) and so we can extract them almost in the same way as we do with the ICs that were stored by the DBMS. However, in the case of being undesirable to put a new table in the database just for the purpose of storing new ICs, the user may choose not to do it in the application interface.

At last, before we get the results of the possible repairs, it is important to check first if the database is already consistent with the previously defined ICs, avoiding unnecessary computations. So, it is also generated a program where all the tuples and the defined

constraints are presented but not the generating rules since we are just checking if the tuples respect the defined ICs. If we get a model resulting from that program then the database is already consistent and we do not need to determine the possible repairs.

6.4 Optimizations

There are several optimizations where the main goal is to minimize the number of groundings and the volume of data to be processed in ASP. The optimizations are the following:

- Avoiding to generate the code of unnecessary tables: In order to reduce the generated ASP code, we only consider relevant tables. Relevant tables are determined by the tables that are present in at least one constraint (defined in the application or previously defined in the DBMS). The list of relevant tables is updated every time the user returns to the ICs definition screen because some ICs may have been added or removed and consequently the relevant tables change. These tables are used to filter tuples that appear in ASP program, the generating of auxiliary rules and the generating rules in order to avoid using code of unnecessary tables, reducing grounding and consequently the computation time;

- Avoiding requests to the DBMS: In order to avoid to constantly request data to the DBMS, that is a slow operation, the database tuples are retrieved when the application starts and stored in a data structure, and then along with the use of the application, they are filtered based on the relevant tables. The ICs defined in the DBMS are also obtained in the beginning of the execution and stored in the correspondent structures to avoid to constantly repeating the operation of getting and processing them;

- Reducing unnecessary attributes instantiation: Not all the attributes are necessary when checking for IC consistency or pruning the answers sets. By knowing it, we replace the unnecessary attributes with a "_", that is, to ignore those values and so, they are not instantiated, reducing the computational complexity. The attributes to be ignored are the ones that are not used in any IC;

- Reducing instantiation on ICs: The rules that form the ICs that prune the answer sets are formulated in order to reduce the number of predicates to be instantiated. That is, it is avoided to define rules that make use of another rules that would cause more instantiation. Each IC, depending on the corresponding semantics, has different "forms" in order to avoid that;

- Reducing the number of generated p_new predicates: We can generate the tuples to be inserted in three different ways. Two of those three involve to determine the combination of several values and each combination forms a tuple. However, that would increase a lot the computational complexity and so, we only generate

the tuples that are useful in satisfying a IC that we are trying to fix. In order to achieve that, when we have ICs where it is mandatory to have tuples with a specific attribute value, we just generate the tuples with those values in the corresponding attributes and the combination of values is just made with the other not mandatory values;

- Avoiding to generate rules related to not allowed operations: For each relevant table, we have to generate the delete, insert and update rules and so if we have a lot of relevant tables, we would have to generate a lot of rules. Having that in consideration, and by knowing that we allow the user to pick what are the allowed operations, we avoid to generate the rules associated with the operations that are not allowed by the user.

6.5 Some examples and analysis of the results

In order to check if the application really returns what is expected we again refer some examples, mention what are the expected results and then show the results obtained by the application.

First, we begin with the Example 7 and the IC that is a foreign key. If we choose to use simple match, the expected results are 1) the removal of the first tuple of the $Students$ table 2) the insertion of a new tuple in $Courses$ table where the $Course$ attribute has the " $AverageCourse$ " value, the $School$ attribute has the " $SchoolA$ " value and the $MaxStudents$ attribute can have a value of the ones defined by the user or a null value and 3) the update of at least one of the referring attributes of the first tuple to null.

In this case the application returns the following:

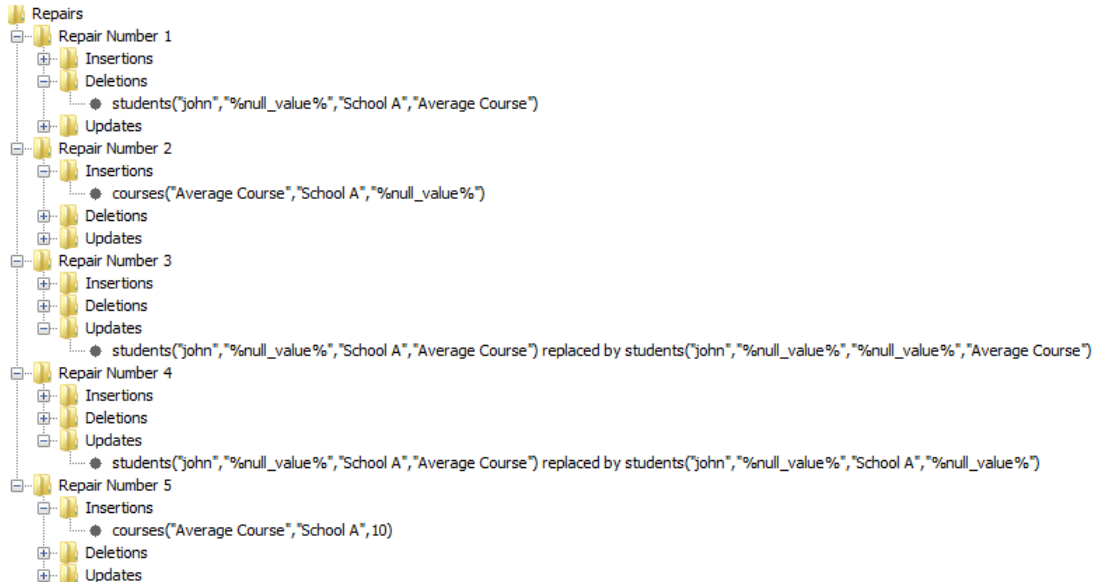

Figure 6.13: Results of foreign key example with simple match

The image shows that the repair number 1 shows the 1) alternative just mentioned.

The repair number 2 and 5 show the 2) alternative mentioned. Here, we just have one repair with a value of the domain because the user just defined that value, otherwise, we would have a different repair for each possible value. The repairs number 3 and 4 show the updates to null combinations on the relevant attributes (the referring ones).

If we choose to use partial match, the expected results for solving the problem with the first tuple of *Students* table is 1) the removal of this first tuple 2) the insertion of a new tuple in *Courses* table where the *Course* attribute has the " *AverageCourse* " value, the *School* attribute has the " *SchoolA* " value and the *MaxStudents* attribute can have a value of the ones defined by the user or a null value. For solving the problem with the second tuple of *Students* table, the expected result is 1) the removal of this second tuple 2) the insertion of a new tuple in *Courses* table where the *Course* attribute has the " *BasicCourse* " value, the *School* attribute can have a value of the ones defined by the user or a null value and the *MaxStudents* attribute can have a value of the ones defined by the user or a null value.

In this case the application returns the following:

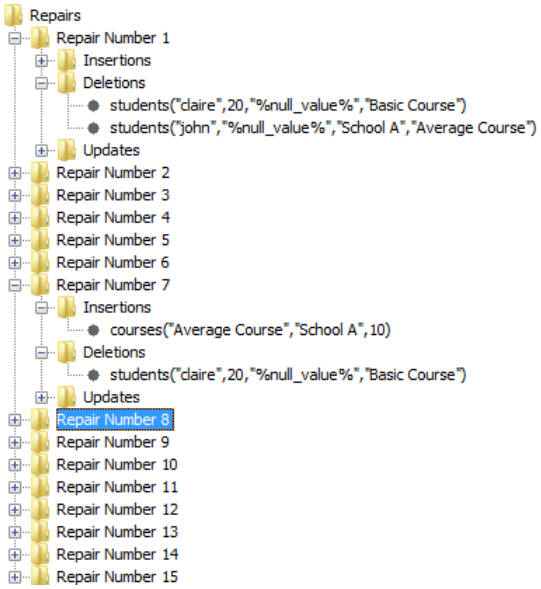

Figure 6.14: Results of foreign key example with partial match

The images show some of the possible repairs. Each repair corresponds to a combination of operations from all the possible combinations that satisfy the ICs. So, they are the combination of the possible alternatives for the first tuple and the possible alternatives for the second tuple together. We can have two insert operations each one solving the problem of each tuple, or two delete operations that delete the problematic tuples, the deletion of a tuple and the update of the other, or the insertion of one tuple and the update of the other. Besides, we also have the combinations of the domain values and the null values when we can apply them.

If we choose to use full match, the expected result for solving the problem with the first tuple of *Students* table is 1) the removal of this first tuple 2) the insertion of a new

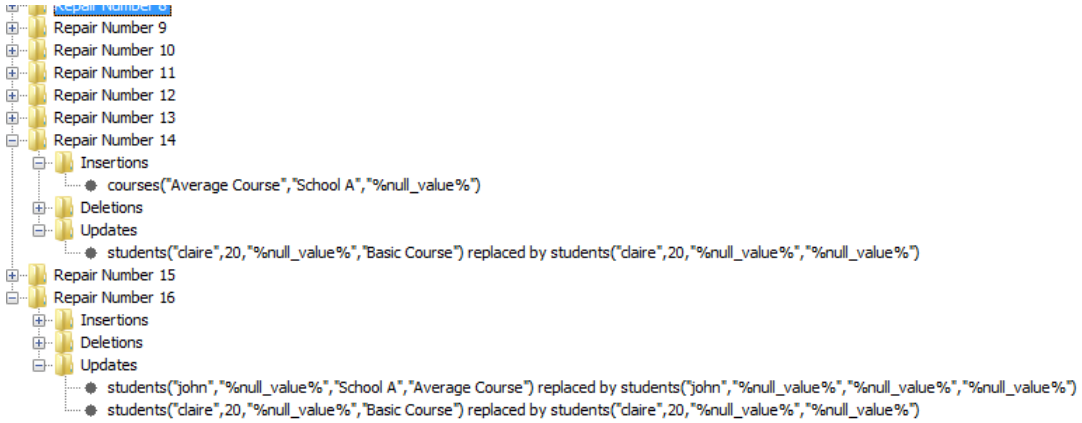

Figure 6.15: Results of foreign key example with partial match

tuple in *Courses* table where the *Course* attribute has the " *AverageCourse* " value, the *School* attribute has the " *SchoolA* " value and the *MaxStudents* attribute can has a value of the ones defined by the user or a null value. For solving the problem with the second tuple of *Students* table, the expected result is 1) the removal of this second tuple. For solving the problem with the third tuple of *Students* table, the expected result is 1) the removal of this third tuple.

In this case the application returns the following:

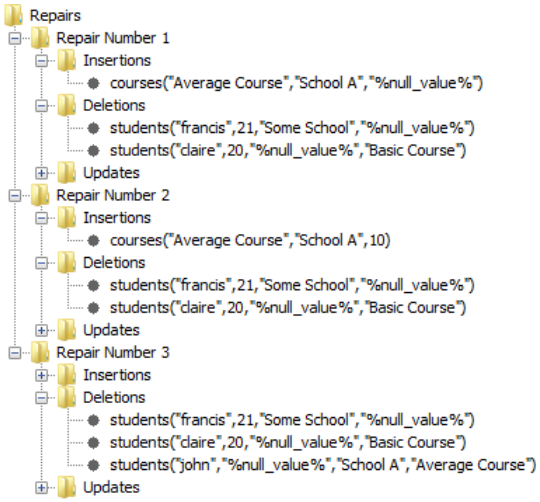

Figure 6.16: Results of foreign key example with full match

Here, since the only alternative for the second and third tuple is to eliminate them, the results show that every repair includes their removal and one of the alternatives of repairing the first tuple. The alternatives of the first tuple include the deletion of the tuple and the insertion of a new one in *Courses* table where we have an attribute that is not a relevant one, so it can be represented with a null value or other value of the domain. Here, we just specified one value for that attribute's domain so it just corresponds to one repair.

Let's look at another example to see if with functional dependencies we get the expected results.

We can look to the Example 11 and the corresponding IC. If we choose to use simple match (or partial match since they have the same behavior as simple match in this case), then we expect the result is that the database is already consistent. In this case the application returns the following:

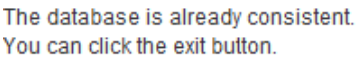

Figure 6.17: Results of functional dependency example with simple match and partial match

If we choose full match then we expect that 1) both tuples are deleted. In this case the application returns the following:

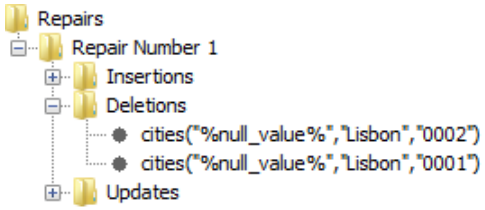

Figure 6.18: Results of functional dependency example with full match

In the image we can easily see that the only possible repair consists on removing both the tuples because both have a null value in one of the relevant attributes.

Let's look at another example to see if with check constraints we get the expected results.

We can look to the Example 13 and the corresponding IC. If we choose to use simple match then we expect that the result is that the database is already consistent. In this case the application returns the following:

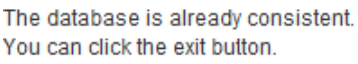

Figure 6.19: Results of check constraint example with simple match and partial match

If we choose to use partial match then we expect that the result is that the database is already consistent too, so the result is the same as in the previous case.

If we choose to use full match then we expect that the result is 1) removal of the second tuple.

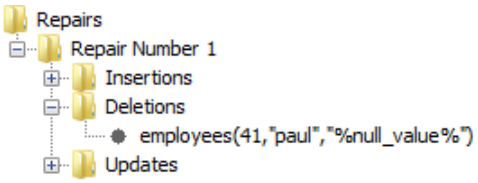

Figure 6.20: Results of check constraint example with full match

In the image we can easily see that the only possible repair consists on removing the second tuple because it has a null value in a relevant attribute.

The final example is to show how the weight minimality works and if the obtained results match with the expected ones.

We use the Example 7 and the IC that is a foreign key, again, with partial-match. If we choose to use the weight minimality criteria with default weight values, the expected results should contain mostly update operations, because we assign a lower weight value for update operations than for insert and delete operations, as we explained before.

For this example, the alternative repairs for the tuples are exactly the same that we mentioned before but this time the minimality is concerned with having more update operations than delete and insert ones because we only need a single update to the first tuple (that is, a weight of $\frac{1}{4}$ instead of deleting or inserting that have a weight of 1) and two updates for the second tuple. So, the obtained results are:

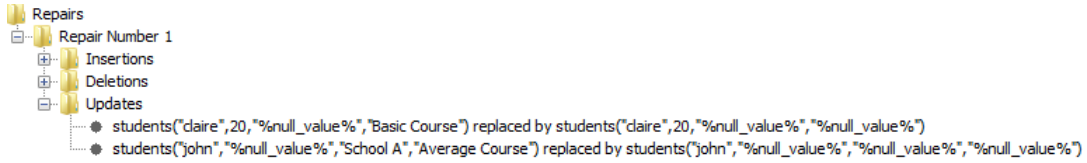

Figure 6.21: Results of foreign key example using weight minimality with default values

The image shows that we get one repair, that only contains update operations and also just the necessary updates for the first and second tuple, as we stated before. As we can see, to repair the database, we only need to update one or two attributes of each corresponding tuple instead of deleting the whole tuple, which means a higher loss of information.

So, when the priority is to reduce data loss, repairing databases with update operations probably is a better option, since they change the least data possible in order to repair the database and, in the worst case, it would change as much data as a deletion operation, when it is really necessary.

However, we can choose to assign new weight values to the operations that we test with the same example again.

Now, we give to the insert and update operations a weight with value 3 and delete operations with value 1. This time, the expected results should contain mostly delete operations.

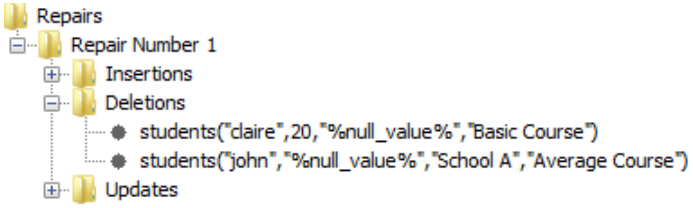

Figure 6.22: Results of foreign key example using weight minimality with custom values

The image shows that we get a repair that only contains delete operations (one for each problematic tuple) as we expected.

With these examples, we conclude that the application returns the expected results for some of the analyzed ICs. However, in these examples, we just used simple and default options in order to easily understand the results of the application and match with the real results.

7

Scalability tests

In Chapter 5, we theoretically evaluated the application where we proved that the results of the ASP code match with the defined theory. In the previous chapter we introduced the developed application, namely its architecture, its features, how to interact with it and also how the output matches the expected results of the repairing process. That is, we already proved in a theoretical and practical way, that the application gives the expected results.

Now, that we already know that the results match with the expecting ones, we need to evaluate the application by its scalability. That is, how the application behaves when dealing with high volumes of information that in this case means the number of tables, the number of tuples, the number of ICs, and so on.

In order to test this, it is necessary to use a realistic database where it is possible to vary the observed characteristics so, we chose to use the TPC-W Benchmark's database. TPC-W[1] is a transactional web benchmark. Its data is based on an "internet commerce environment that simulates the activities of a business oriented transactional web server", it models an online bookstore. It is structured as in the following database schema[2]:

[1] TPC-W Benchmark homepage: http://www.tpc.org/tpcw
[2] Generated using DbVisualizer 9.0.8

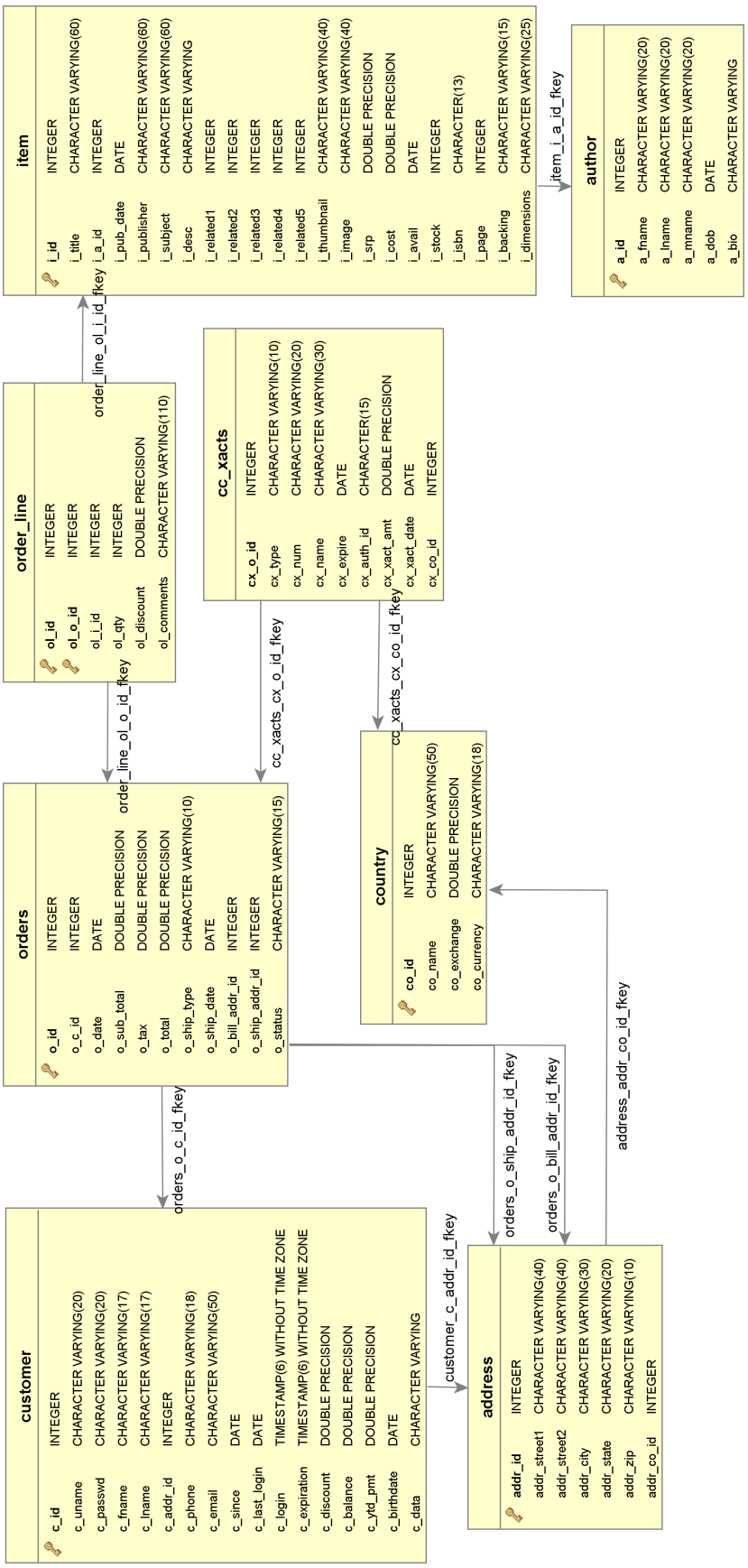

Figure 7.1: TPCW database schema

The other advantage of using this database, besides its dimension, is the possibility of comparing the results of the experiments with the ones performed in [Alv11]. We actually use the database and tests instances that were used in [Alv11] and that are available online [3].

Therefore, we decided to analyze the influence of some characteristics that are analyzed in [Alv11] so that we can compare them and some other characteristics that are more specific to the application developed in this dissertation. Then, the analyzed characteristics are the following:

1. Number of relevant tables considered;
2. Number of irrelevant tables considered;
3. Number of new integrity constraints considered;
4. Number of new irrelevant integrity constraints considered;
5. Tables operations limit defined by the user;
6. Number of relevant attributes that can be updated for repairing purposes;
7. Obtaining repairs only with null values confronted with repairs without null values;
8. Number of attributes that compose the tables involved in the repairing process;
9. Number of tuples of the tables that are involved in the repairing process.

All the tests are performed in a AMD Athlon^{TM} X2 Dual-Core QL-62 @ 2.00 GHz with 6 GB RAM, with Windows 8 and 64 bits operating system and the tests of [Alv11] were performed in an Intel ® Core ^{TM} i7 CPU 920 @ 267 GHz with 6 GB of RAM, with Windows 7 Professional and 64 bits operating system. Both use PostgreSQL 8.4, we use Gringo version 3.0.5 instead of version 3.0.3 that is used in [Alv11], claspD version 1.1.2 and clasp version 2.1.1. instead of 1.3.4 that is used in [Alv11]. Taking this in consideration, the performances of the computers do not completely match. However, we try to compare the type of behavior took by the applications in each test.

Each test was repeated three times and the results presented here are the average of these three so that we do not provide results that represent deviations. Also, each result represents the sum of the average of consistency check times and the average of the times of getting the alternative repairing answers.

[3] http://sourceforge.net/projects/drsys/files/drsys1.0

7.1 Tests results and analysis

7.1.1 Number of relevant tables

The tables that are referred in at least one IC are considered to be relevant tables. Those ICs may specify dependencies between tables and so, when we change a table's data from which other depends on, their data may have to be changed too. That is, these dependent tables are also relevant.

For example, considering Figure 7.1, if we consider the $Author$ table as a relevant table, then $Item$ is also a relevant table because the attribute i_a_id refers the attribute a_id from $Author$ table and so, if we delete some tuple from the latter, there may be a tuple from $Item$ table referring it.

In this test we want to notice the influence of the number of relevant tables on application's performance. If a lot of tables depend from a table that participates in an IC, it affects the program execution time? For that, we use a database instance of TPC-W Benchmark provided by [Alv11] that is composed of 10.000 tuples and the following ICs:

$$F_1 = \langle Country, co\_id, \leq, 1\rangle^{ckc}$$
$$F_2 = \langle Author, a\_id, \leq, 1\rangle^{ckc}$$
$$F_3 = \langle Orders, author, \langle o\_date\rangle, \langle a\_id\rangle\rangle^{fk}$$

Where F_1, as explained in [Alv11], means that all tuples from $Country$ table must have a value lower or equal to 1 for co_id attribute, that is, we cannot have tuples such that the value of co_id attribute is bigger than 1. Its relevant tables are: $Country$, $Address$, $Customer$, $Orders$, $Order\_line$ and Cc_xacts . F_2 means that all tuples from $Author$ table must have a value lower or equal to 1 for a_id attribute, that is, we cannot have tuples such that the value of a_id attribute is bigger than 1. Its relevant tables are: $Author$, $Item$ and $Order\_line$. The last IC, the F_3 one, means that every value from $Orders'$ o_date attribute must exist in the values of $author'$s a_id attribute. Its relevant tables are: $Author$, $Item$, $Order\_line$, $Orders$ and Cc_xacts .

In all tests, we included the ICs already defined in the DBMS (primary keys, foreign keys to establish the relation between tables and not-null constraints) and a growing number of relevant tables until including all relevant tables with respect to the analyzed IC. The first test uses F_1; the second one uses F_2; the third one uses F_1 and F_2 and the fourth uses F_3.

About the other options, the minimality criteria chose is the cardinality criteria and the only allowed operations are delete operations.

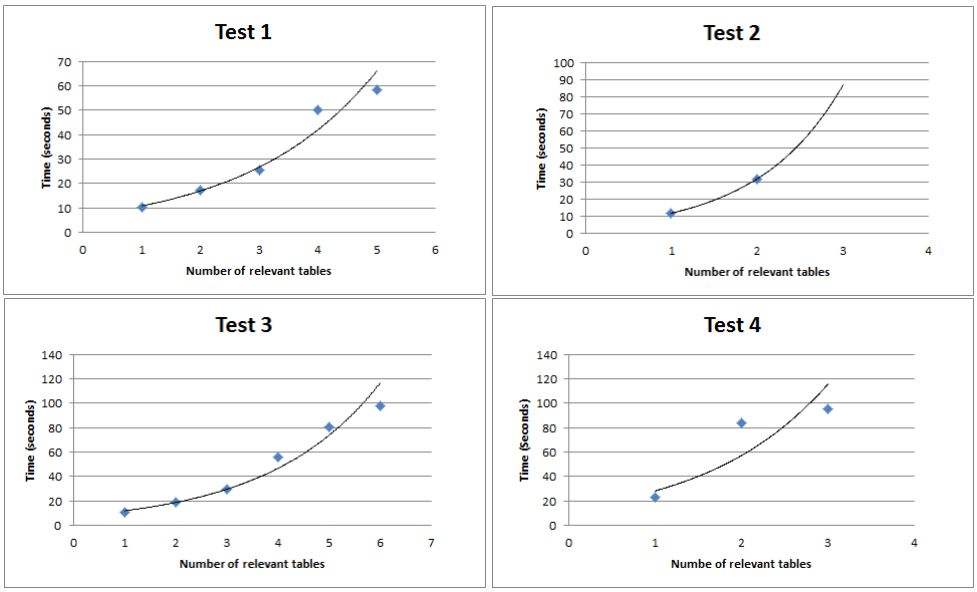

Figure 7.2: Number of relevant tables results

As we can see in the charts, increasing the number of relevant tables affects a lot the execution time. This happens probably because, by considering more tables, we are also increasing the number of considered tuples and the number of generated combinations for deleting or not each tuple. With more tables, there is an explosion on the grounding, and there are more alternative repair combinations that can be generated because there are tables that are dependent from the relevant ones and so they are also repaired if needed.

We can also state, by observing the curves of the charts, that the referred time grows exponentially. For example, if we observe the chart of test 1, we can see that by using 2 relevant tables we get approximately 17 seconds to get the repairs results and when using the double of the tables, that is, 4 tables, we do not get the double of the time but 50 seconds.

About comparing the obtained charts between them, we cannot actually compare them because they are related to different tables and also different ICs so the time values vary.

7.1.2 Number of irrelevant tables

Besides having relevant tables, there are also the irrelevant ones. These are the ones that do not participate in any IC and so, should not be included in the logic program, avoiding unnecessary computations.

For example, considering Figure 7.1, considering that we are defining the IC $\langle Item, i\_cost, >, 0\rangle^{ckc}$, then $Order\_line$ is a relevant table because the attribute ol_i_id refers the attribute i_id from $Item$ table, however, Cc_xacts is an irrelevant table because it does not depend neither from $Item$ nor $Order\_line$ tables, direct or indirectly.

In this test we want to observe the influence of the number of irrelevant tables. Since

they are irrelevant for generating repairs, would they affect the program execution time?

Here, we use a database instance provided by [Alv11] that is composed of 10.000 tuples and the ICs introduced in the last subsection: F_1, F_2 and F_3. The first test uses the IC F_1; the second one uses the IC F_2; the third one uses the ICs F_1 and F_2 and the fourth uses the IC F_3. In all tests we included the ICs already defined in the DBMS (primary keys, foreign keys to establish the relation between tables and not-null constraints) and for each test we include all the relevant tables with respect to the IC that is being tested and a growing number of irrelevant tables until including eight irrelevant tables. All irrelevant tables have four attributes and 1.000 tuples.

About the other options, the minimality criteria chose is the cardinality criteria and the only allowed operations are delete operations.

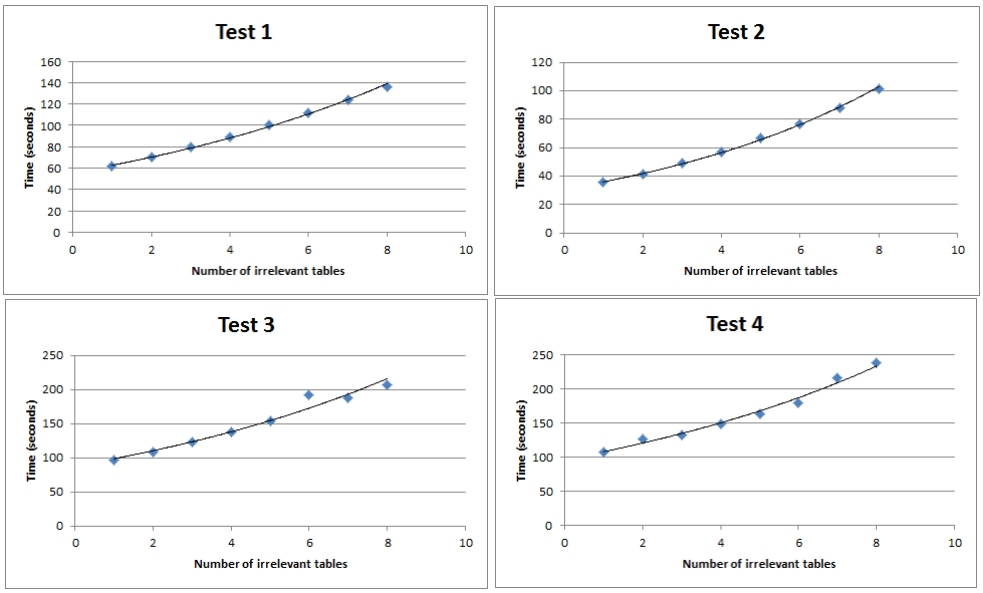

Figure 7.3: Number of irrelevant tables results

As we can see in the charts, increasing the number of irrelevant tables affects the execution time. This happens probably because, by considering more tables, we are also increasing the number of considered tuples and the number of generated combinations for deleting or not each tuple. With more tables, there are more alternative repair combinations that can be generated even when these combinations are not relevant, resulting in more grounding when generating these combinations.

This test was also perform in [Alv11]. There we can observe that the execution time also increases with the increasing of the number of irrelevant tables. However, there, the execution time grows exponentially while here it grows linearly. This can occur because of possible changes between Gringo and clasp versions because here, we use more recent ones. However, it may occur due to other motives related to how the program is built. For example, there, they use Lua to retrieve tuples from the database instead of using Java to retrieve them as we do here. Also, the rules used there lead to more instantiation (e.g. here we just ignore the irrelevant attributes in IC rules but there they make use of

other predicates with different arities to ignore those attributes, leading to more instantiation). Finally, the times are different but in both applications the time increases with the increasing number of tables.

These results show how our application's feature of automatically determining the relevant tables is important. This situation would never occur in the application, since it determines which are the relevant tables to be considered for the logic program based on the tables included in at least one IC. So, it would never consider as relevant a table that does not need to be repaired, avoiding the extra computation.

7.1.3 Number of new integrity constraints

The user can define an unlimited number of ICs, in the application, that are expected to provoke changes on data.

For example, considering Figure 7.1, considering that we are defining the IC $\langle Item, i\_cost, >, 10\rangle^{ckc}$ and that we do a query to the item table:

```
select i_id, i_cost from item
```

And suppose that we get:

i_id	i_cost
1	*12*
2	*9*
3	*7*

Table 7.1: Item table instance

The $Item$ table instance is violating the IC in second and third tuples, that is, we need to repair it, and so, the IC revealed to be relevant.

In this test we want to observe the influence of the number of ICs. If the user defines a lot of ICs, would it affect the program execution time?

Here, we use a database instance provided by [Alv11] that is composed of 5.000 tuples and the following ICs:

$$F_1 = \leftarrow p\_keep_{country}(RowId, Id, \_, \_, \_), p\_keep_{author}(RowId2, Id, \_, \_, \_, \_, \_)$$
$$F_2 = \langle Address, addr\_co\_id, \leq, 25\rangle^{ckc}$$
$$F_3 = \langle Country, co\_id, \leq, 40\rangle^{ckc}$$
$$F_4 = \langle Author, a\_id, \geq, 45\rangle^{ckc}$$

With respect to their meaning, F_1 forbids tuples from $Country$ and $Author$ tables to have the same value for the co_id and a_id attributes, respectively; F_2 forbids tuples from $Address$ table to have values greater than 25 for the $addr\_co\_id$ attribute; F_3 forbids tuples from $Country$ table to have values greater than 40 for the co_id attribute

and F_4 forbids tuples from $Author$ table to have values lower than 45 for the a_id attribute.

In all tests, we include the ICs already defined in the DBMS (primary keys, foreign keys to establish the relation between tables and not-null constraints) and considered relevant tables are all the tables of the database because they all depend from the tables referred in the ICs. The first test uses F_1 first, then F_2, F_3 and finally F_4; the second one uses F_4, F_3, F_2 and finally F_1; and the third one uses F_3, F_2, F_4 and finally F_1.

About the other options, the minimality criteria chose is the cardinality criteria and the only allowed operations are delete operations.

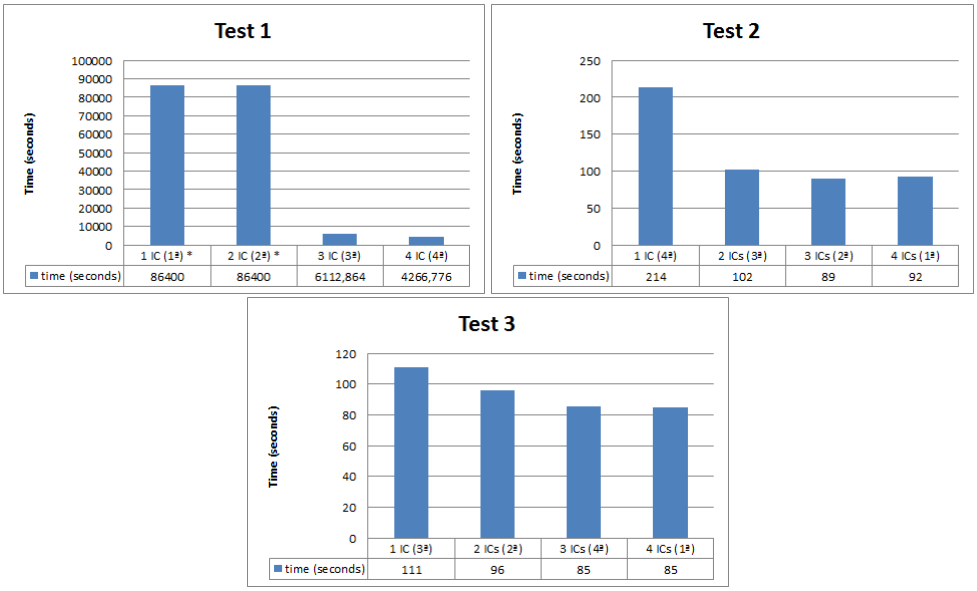

Figure 7.4: Number of relevant ICs results

As we can see in the charts, by adding more ICs, the execution time decreases. The latter is also affected by the ordering of the ICs, as we can see in the tests results, because they all make use of the same ICs but in different orderings which results in very different times. For example, if we add F_1 in first place (test 1), we get a much higher execution time than we get when it is added after the other ICs (test 2 and test 3).

These results are very similar with the results obtained in [Alv11] where the charts are very similar (they have the same shape) even with different time results. This similarity was expected because the logic program generated by both approaches is similar and the ICs have very similar transformations too. But why this happened here and in [Alv11]? By adding constraints we can increase the grounding that is made (because the ICs must be instantiated) but constraints prune answers and it can be significant in reducing the search space. Some ICs prune more answers than others and so, the ordering matters because if we start with an IC that reduces a lot the search space, the following ICs may have less alternatives to deal with and take less time to filter the answers that fit. For example, if we have two ICs that prune a lot of answers from the search space, then by having one after the other, the second one may not prune a significant part of the search

space because the first IC has done it before. So, here, we can state that what matters when increasing the number of ICs is not the number of them, it is the contribute that they have to prune answers. We can justify this behavior with strategies adopted by the solver, clasp, that are also mentioned as a motive in [Alv11].

Clasp allows to apply advanced techniques from Constraint Processing and Satisfiability Checking as backjumping and conflict-driven learning, making use of an approach centered in the concept of a nogood. A nogood is a set of literals that results in a constraint violation and that can be learned from conflicts [GT09]. They are preprocessed using techniques adapted from Satisfiability Checking [GKS09] and are checked when computing answer sets. Then, we can state that when there is a conflict, clasp saves the conflict (a constraint) expressed as a nogood, and when an assignment violates some nogood then it is not extended to a solution [GT09] and so, by rejecting possible solutions, it is restricting the search space and saving time.

Note: The executions that are signed with a "*" do not ended until reach 86.400 seconds but we decided to end it because it is not meaningful to let it run more time.

7.1.4 Number of new irrelevant integrity constraints

As mentioned before, the user can define an unlimited number of ICs. However, not every IC actually provokes changes on data and in that case, it makes no difference when they are or not defined because the database is not repaired with respect to that IC.

For example, considering Figure 7.1, considering that we are defining the IC $\langle Country, \{co\_id\}\rangle^{pk}$ and that we do a query to the $Country$ table:

```
select co_id from country
```

And suppose that we get:

co_id
1
2
5

Table 7.2: Country table instance

The $Country$ table instance is not violating the IC because the values for the co_id are already unique, without repetitions and also do not have null values and so, all the constraints for defining a primary key in that attribute are already accomplished.

In this test we want to observe the influence of that number of irrelevant ICs. If the defined ICs are irrelevant, would it affect the program execution time?

For this test, we use a database instance provided by [Alv11] that is composed of 5.000 tuples and the following ICs:

$$F_1 = \langle Country, co\_id, \leq, 40\rangle^{ckc}$$
$$F_2 = \langle Address, addr\_id, \geq, 45\rangle^{ckc}$$

Where F_1 forbids tuples from $Country$ table to have values greater than 40 for the co_id attribute and F_2 forbids tuples from $Address$ table to have values lower than 45 for the $addr\_id$ attribute. With respect to those ICs, the considered relevant tables for F_1 are: $Country$, $Address$, $Customer$, $Orders$, $Order\_line$ and Cc_xacts . The considered relevant tables for F_2 are: $Address$, $Customer$, $Orders$, $Order\_line$ and Cc_xacts .

About the tests, for the first test we use F_1 and, first, we execute the program considering all primary key constraints associated with the considered relations and then without considering them. For the second test we use F_2 and then, first, we execute the program considering all primary key constraints associated with the considered relations and then without considering them. For both tests the primary keys are irrelevant ICs and we include the ICs already defined in the DBMS (primary keys, foreign keys to establish the relation between tables and not-null constraints) .

About the other options, the minimality criteria chose is the cardinality criteria and the only allowed operations are delete operations.

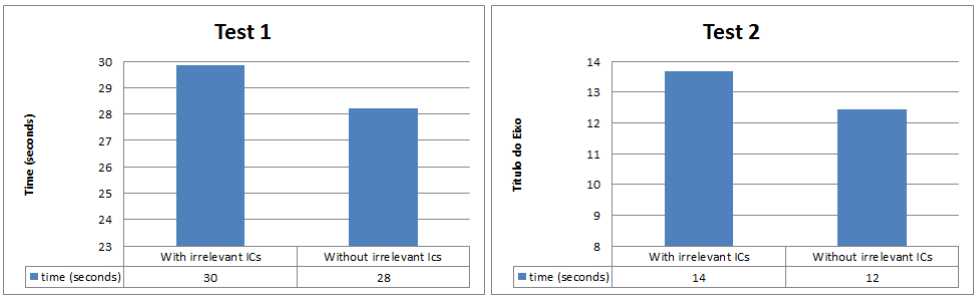

Figure 7.5: Number of irrelevant ICs results

As we can see in the charts, by increasing the number of irrelevant ICs, the execution time increases a little but it is not significant so, we can conclude that the number of irrelevant ICs does not affect the performance of the application.

These results are very similar with the results obtained in [Alv11] where the charts have the same shape, the same behavior. This similarity was expected because the ICs have similar transformations in both approaches and also because both use the same solving mechanisms when using clasp. We can justify this indifference in using irrelevant ICs or not through Gringo and clasp mechanisms. As we can see, by using irrelevant ICs, we have a higher time that when not considering them. This happens because, when considering them, we need to do more grounding to instantiate the ICs, and as they do not prune answers (since they are irrelevant) then the time does not decreases. However, when not considering them, we avoid the grounding, reducing the time.

As mentioned before, clasp makes use of nogoods to record conflicts and reducing the search space. However, these irrelevant ICs are not violated by any instance of the database (they do not have corresponding nogoods) and so, they do not represent conflicts, that is, they can be ignored, as stated in [Alv11], not producing any modification in performance.

7.1.5 Tables operations limit

In order to repair a database we can use a set of delete, insert or update operations. However, the user can define a limit to the number of operations that can be made to a specific table.

For example, considering Figure 7.1, considering that we are defining the IC $\langle Order\_line, ol\_discount, \leq, 20\rangle^{ckc}$ and that we do a query to the $Order\_line$ table:

```
select ol_id, ol_discount from order_line
```

And suppose that we get:

ol_id	$ol\_discount$
1	*35*
2	*12*
5	*21*

Table 7.3: Order_line table instance

The $Order\_line$ table instance is violating the IC because of the first and third tuples. Suppose also that we establish a limit of one delete operation for $Order\_line$ table. We cannot repair the database deleting the first and third tuples because of the limit for that table is of one delete operation but we can update the attribute $ol\_discount$ to $null$ in both tuples without disrespect any constraint.

In this test we want to observe the influence of that number of relation operations limit. If we define a low maximum number of operations, would it affect the program execution time? Would it take more or less time with a high maximum number of operations?

For this test, we use a database instance provided by [Alv11] that is composed of 15.000 tuples and the following IC:

$$F = \leftarrow p\_keep_{order\_line}(RowId, \_, \_, \_, Ol\_qty, Ol\_discount, \_),$$
$$p\_keep_{order\_line}(RowId2, \_, \_, \_, Ol\_qty, Ol\_discount2,), Ol\_discount < Ol\_discount2.$$

F forbids two tuples from $Order\_line$ table to have the same values for ol_qty attribute, being the $ol\_discount$ attribute of one tuple lower than the same attribute of the other tuple. Considering this IC, the only relevant table is the $Order\_line$ table.

Finally, about the tests, they consider the constraint F and the execution is repeated with different limits for the number of delete operations for the $Order\_line$ table, that are: 255 (corresponds to the minimal repair), 300, 400, 500 and unbounded (without limit). Also, in all executions, we include the ICs already defined in the DBMS (primary keys, foreign keys to establish the relation between tables and not-null constraints) .

About the other options, the minimality criteria chose is the cardinality criteria and the only allowed operations are delete operations.

Figure 7.6: Limited number of operations results

As we can see in the chart, by increasing the limit of allowed operations, the execution time does not represent big changes. This is because the only answer that is returned is the minimal answer, that is, that corresponds to the 255 limit. So, it makes no indifference to use them or not because they do not reduce the execution time. Otherwise, if we had other answers composed of more operations than the minimal answer, the limits could probably reduce the search space and consequently reduce the execution time. However, the result that considers the 255 value should result in an equal/very similar time with respect to the other limits since the answer that is found is the same one. The justification for this is not understood, so it probably is due to the internal functioning of clasp.

In [Alv11] the time decreases when the limit is closer to the minimal repair because in their approach, the number of operations is already limiting the generating of operations predicates thus, never generating answers where the limit is exceeded and so the search space is smaller when the limit is also smaller.

It is also important to note that if we specify a limit that is lower than the number of operations that compose the minimal answer, then we would not get any repairs.

7.1.6 Number of relevant updatable attributes

Instead of deleting or inserting whole tuples, we provide the feature of updating specific relevant attributes to $null$.

For example, if we consider the example of the previous subsection, the $Address$ table instance is violating the IC because of the first tuple. Instead of generating and inserting new tuples, we can just update the $address\_co\_id$ to $null$ and the database is repaired.

In this test we want to observe the influence of that number of attributes that are possible to be updated. If we allow to update a big number of attributes to $null$, would it affect the program execution time?

For this test, we use a database instance provided by [Alv11] that is composed of 5.000 tuples and the following ICs:

$$F_1 = \langle Orders, Customer, \langle o\_c\_id \rangle, \langle c\_id \rangle \rangle^{fk}$$

$$F_2 = \langle Orders, Address, \langle o\_bill\_addr\_id \rangle, \langle addr\_id \rangle \rangle^{fk}$$
$$F_3 = \langle Orders, Address, \langle o\_ship\_addr\_id \rangle, \langle addr\_id \rangle \rangle^{fk}$$

The ICs state that for every tuple that belongs to $Orders$ table, the referred attributes must exist in $Customer$ table (in the case of o_c_id attribute) and exist in $Address$ table (in the case of $o\_bill\_addr\_id$ and $o\_ship\_addr\_id$ attributes). The considered relevant tables are $Orders$, $Customer$ and $Address$.

Finally, all tests use the three ICs but first, not allowing updates in any of the referring attributes, then allowing only to update one attribute, then allowing two and then three attributes. Also, in this test, we do not include the ICs already defined in the DBMS (primary keys, foreign keys to establish the relation between tables and not-null constraints).

About the other options, the minimality criteria chose is the cardinality criteria and the only allowed operations are delete and update operations.

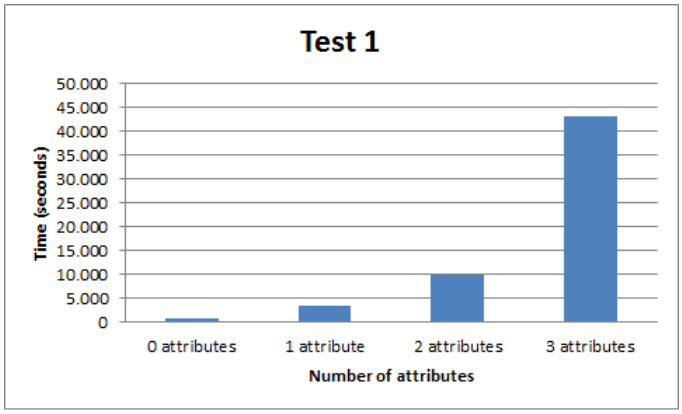

Figure 7.7: Number of updatable attributes results

As we can see in the chart, increasing the number of updatable attributes changes a lot the execution time and, as in previous test, even with a small number of values. This has a reason similar to the previous test because, when we allow to update the tuples of a table, it is generated the combination of possible updates to each relevant attribute of each tuple of the table that is being updated, which is a complex computation because we are increasing a lot the number of combinations that are done that can possibly repair the database.

We can also state, by observing the bars of the charts, that the time growth does not scales very well, because, for example, if we choose to only update one attribute we waste around 3.000 seconds and with the double of the attributes, that are only two, we waste not the double of the time but around 10.000 seconds.

7.1.7 Obtaining repairs only with null values vs without null values

As noticed in the previous test, updates are heavy operations because of the combinations that are necessary to do to generate the multiple alternatives. The user can choose to wait and be able to choose the combination of values that compose the tuples that she wants to

be inserted or make use of null values and instead of getting the combination of values, use null values when possible.

For example, returning to example of Number of domain's values subsection, instead of generating the tuples like $Country(4, UnitedStates, 2, Dollars)$ or $Country(4, UnitedStates, 1, Dollars)$, we can just use $Country(4, null, null, null)$.

In this test we want to observe the influence of using or not-null values instead of generating the values combinations when inserting new tuples. If we choose to use null values instead of regular values for insertion purposes, would it affect the program execution time?

For this test, we use a database instance provided by [Alv11] that is composed of 10.000 tuples and the following IC:

$$F = \langle Address, Country, \langle addr\_co\_id \rangle, \langle co\_id \rangle \rangle^{fk}$$

F states that for every tuple that belongs to $Address$ table, the referred attribute must exist in $Country$ table. The considered relevant tables are $Address$ and $Country$.

Finally, about the test, we first run the program using only null values when doing insertions and then we run it again but using some combinations of values. Also, in this test, we include the ICs already defined in the DBMS (primary keys, foreign keys to establish the relation between tables and not-null constraints).

About the other options, the minimality criteria chose is the cardinality criteria and the only allowed operations are delete and insert operations.

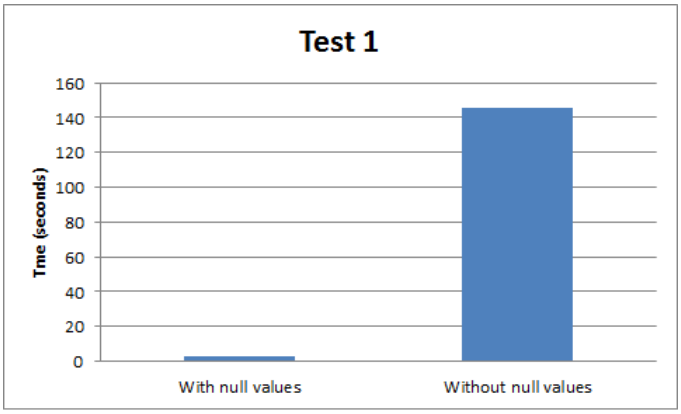

Figure 7.8: Influence of using null values or not results

As we can see in the chart, it takes much more execution time to use the combination of values than to choose to use only null values. As stated before, the computation of the combinations is a very heavy approach and so, taking advantage of using null values to replace the multiple alternative values is a good option, because it avoids to compute combinations, reducing the grounding.

7.1.8 Number of attributes that compose the tables

We already mentioned the influence of some parameters that are specified by the user as the defined ICs and the maximum number of operations. Now, we analyze the influence of one of the characteristics that belong to the structure of the database, that is, the number of attributes.

For example, considering Figure 7.1, considering the $Item$ table. Since it has 22 attributes, would it be relevant if instead of 22 it had 10 attributes, for example?

In this test we want to observe the influence of that number of attributes in a table that belongs to some IC.

For this test, we use a database instance provided by [Alv11] that is composed of 5.000 tuples and the following IC:

$$F = \langle Item, i\_cost, \leq, 30000 \rangle^{ckc}$$

F forbids tuples from $Item$ table to have values greater than 30.000 for the i_cost attribute. Considering F, the relevant tables are: $Item$ and $Order\_line$.

Finally, about the test, it is repeated four times, where in each time we consider that the $Item$ table (the table directly covered by the IC) has a different number of attributes. Also, in all tests, we include the ICs already defined in the DBMS (primary keys, foreign keys to establish the relation between tables and not-null constraints).

About the other options, the minimality criteria chose is the cardinality criteria and the only allowed operations are delete operations.

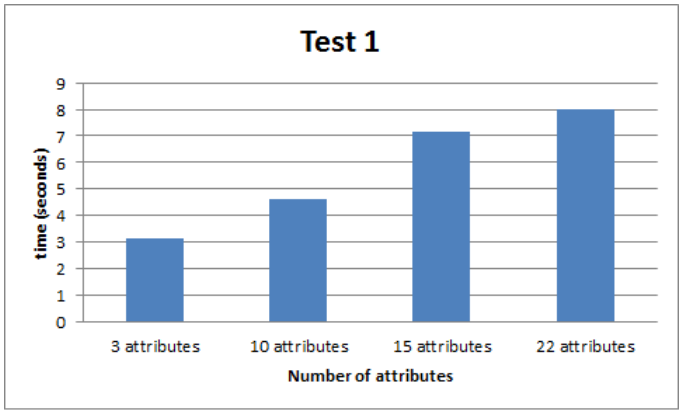

Figure 7.9: Number of attributes results

As we can see in the chart, by increasing the number of attributes, the execution time increases a little but not with very relevant values. The execution time increases because with more attributes it is necessary to do more grounding however, the program is optimized to avoid irrelevant grounding in ICs rules and the grounding does not participate in the more relevant part of the computation that corresponds to generating the multiple combination of operations to perform to the database. We can then conclude that the number of attributes does not affect the performance of the application.

7.1.9 Number of tuples of the tables

The tables that are referred in the ICs can have a lot of tuples or a small number of them.

For example, considering Figure 7.1, considering the $Address$ table. Would it be relevant if it has 1.000 tuples or 100 tuples, for example?

In this test we want to observe the influence of that number of tuples in a table that belongs to some IC.

For this test, we use a database instance provided by [Alv11] that is composed of 10.000 tuples and the following IC:

$$F = \langle Order\_line, ol\_discount, \leq, 26 \rangle^{ckc}$$

F forbids tuples from $Order\_line$ table to have values greater than 26 for the $ol\_discount$ attribute. Considering F, the only relevant table is $Order\_line$ table.

Finally, about the tests, the repairing process is repeated with different number of tuples in the table where we define a new IC. First, with 20 tuples, then 100, 300, 500, 800 and 1.500. Also, we included the ICs already defined in the DBMS (primary keys, foreign keys to establish the relation between tables and not-null constraints).

About the other options, the minimality criteria chose is the cardinality criteria and the only allowed operations are delete operations.

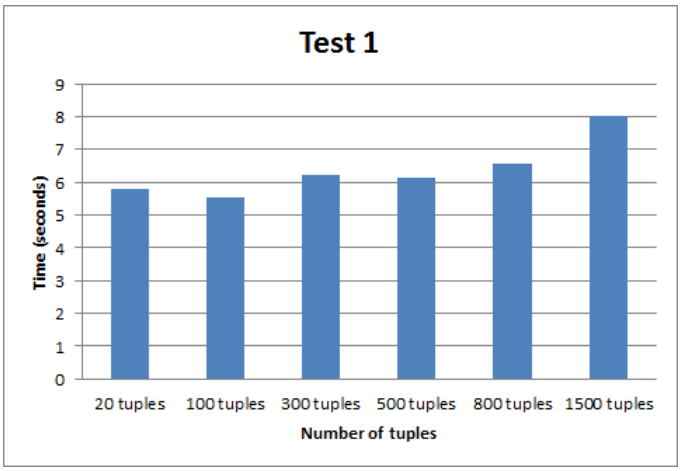

Figure 7.10: Number of tuples results

As we can see in the chart, by increasing the number of tuples, the execution time increases a little but it is not very accentuated as the number of tuples increases. The time increases because there are more tuples to check and more tuples that can represent inconsistencies and consequently more tuples to repair. Probably, it does not increases the time a lot at each tested number of tuples because that number does not represent more combinations of operations, which is more complex than just having more grounding, that is what happens here. Also, sometimes when doing more grounding, it can result in a less computation time due to intelligent grounding that avoids to ground the instances that are not relevant.

7.2 Overall considerations

Overall, by increasing the volume of handled data, we are increasing the execution time of the application. This generally occurs because we are increasing the grounding of the program, because with bigger programs we get more grounding since there are more variables to instantiate and more combinations to generate. This not always occur because of intelligent grounding, where sometimes, it is more efficient with more grounding because it avoids the instantiation of variables that are pruned later for some constraint.

What also influences the performance of the program is the ICs that are used. If we provide features that help to restrict the search and provide constraints that prune a lot of the search space, then it means a big improving on the performance, as we witnessed in the previous example.

Also, what gives "weight" to the program is the solving part because it has to deal with lots of combinations and consequently, alternative repairs and so, it takes a big piece of the execution time.

One limitation noted during the tests phase, was the memory RAM available in the machine where the tests are performed that establishes a limit for the handled data and when exceeded, it causes a program's crash.

8

Conclusion

In this dissertation we analyzed the problems of considering null values, noticed the multiple interpretations that a null can has, analyzed the existing semantics to deal with their ambiguous meaning and proposed a solution for dealing with them in constraint satisfaction and repairing process.

Developments

One of the main components of this dissertation is composed by the study of the state of the art that allowed to analyze the multiple approaches available to deal with null values and providing examples to each IC type according the semantics. The study provided in this document complements what is already available in the state of the art because they provide multiple semantics but do not do a diagnosis of each one of them in the context of each different IC. Also, we suggest a best semantics to each IC and formalize it.

About the repairing part, we determined how to repair the databases that represent inconsistencies regarding the previously analyzed ICs as the allowed operations, where we decided to allow insertions and deletions in order to avoid the database to just grow or just to shrink when repaired. We also decided to allow updates to null values because avoids to lose a lot of information when deletion is the only choice. Also, updates are operations that may consider less lost of information considering delete operations because, as opposed to delete operations, it may consist of updating less attributes that the total of attributes of a specific table. These considered tables are not all the tables of the database, they are filtered by the application to do not being considered tables that do not influence the repairing process, that is, the ones that do not appear in the ICs and that are not related with the tables that participate in the ICs. This is a contribution to the

performance of the application, as analyzed in Chapter 7.

About the minimal answers, we decided to allow cardinality, under-set inclusion and weight minimality in order to cover the majority of cases. Regarding weight minimality, it presents advantages when preferring some operations over the others. For example, we may prefer to reduce the size of the database instead of increasing it, that is, preferring delete operations over insert operations. This criteria also provides a way of preferring update operations because they reduce the amount of lost information, as just mentioned.

About the implementation, which is one of the big contributions of this dissertation, since there are not available applications that repair databases with null values, besides allowing to repair a database with null values, it allows to specify which is the semantics to apply to each of the specified IC in the moment of the specification. Also, it allows to specify multiple options regarding the allowed operations. For example, it allows to specify tables from where is not possible to remove tuples, tuples that cannot be updated or the maximum number of insert, delete or update operations. Also a valuable feature is one where it is possible to choose what kind of answers to determine: repairs that make use of null values to generate tuples to be inserted or updated, or repairs that do not consider null values for repairing proposes. As seen in Chapter 7, this has advantages in the performance of the application.

In this Chapter 7, we also showed the influence of some evaluated parameters as the influence of the number of tables, the type of answers obtained, the number of updatable attributes, and showed how the grounder and the solver behave to provide the best performance possible.

We can now compare our decisions with the approach that, as mentioned, is very similar to this one when not considering null values, [Alv11]. There it is defined the same type of ICs that we define in the dissertation and we also provide basically the same options related to operations. However, we have the advantage of providing two more options to ease the process of generating new tuples based on specified attribute domains. Also, in this approach the authors use the same minimal criteria as we do, use DR instead of CQA and also implement the application in ASP. However, null values are excluded from the domain of the attributes and even do not insert them, it was left as future work, and we do not provide the feature of specifying ICs using SQL because we do not think that this is a very important feature because the user can already specify them using the interface or ASP.

Limitations

After all the developments just mentioned, that are also some points that could be improved.

Considering the development of the application, about the ICs, the check constraints are a little limited because they only allow to refer attributes of a single table for the same IC. There are also some features that may have been provided as, for example, the update

of null values to regular values, because we already have the update of regular values to null values and the opposite may also be useful. Also about the updates, it could be useful to choose not to update specific attributes, as we allow to choose not to update specific tuples.

However, the main limitation of the application is the fact that it takes a lot of time to compute the repairs for large databases. This is a consequence of the application just getting the repairs that are absolutely correct and precise, not approximations, as we proved with the soundness and completeness proofs. It is a complex problem. If the user prefers to use an application with approximate repairs but with a better performance, then this application would also be useful to compare the results and for benchmark tests to more efficient solutions.

Future Work

Taking into account the previous limitations, there are some improvements that are left for future work. About the theoretical part, there are some improvements that can be made and that were just mentioned, but now we focus on improvements on the developed application.

About the application interface, that was made in Java, it would be nice to have a more user-friendly one making use of for example, some drag and drop to put the tables and attributes names in the corresponding boxes. At the moment, Java is used to handle the interaction with the user which means that the computer where the application runs has to have Java installed. In order to overcome this disadvantage and also to make the application available in every place and every computer, it could be developed a web version of the application, which also would contribute to a more user-friendly interface, certainly.

Something that was already mentioned is about saving ICs. It could be improved by, for example, saving the ICs in a file or even in the original database but without participating in database's schema.

Finally, it was noticed, when performing scalability tests, that in some cases the time to wait for alternative repairs is a little long. In the application, it happens that the user has to wait for all repairs to be obtained and then choose one of the alternative repairs. In order to avoid the user to wait for all the answers, it could be a good improvement to show the answers to the user as they are obtained from the solver and so the user has to wait less time to start to see some answers. In the case of the user wants to see all the repairs, then she can wait until the process is terminated.

To speed up the process, it could also be interesting to use concurrency as suggested in [Alv11]. The goal here is to split the work by a set of nodes. However, we have to pay attention in how to split the work so that we do not introduce possible inconsistencies. For example, we should not split a single table where we apply an IC by multiple nodes because, if the IC needs to compare the tuples with each other, each node may consider

some tuples as consistent without comparing with the tuples from the other nodes, and so, in the end when the results are merged, we still have inconsistencies. We could also split the ICs by different nodes but, as noticed before, have many ICs is good for improving performance, and in the case of splitting ICs that refer the same table, we are worsen the performance in each node. A good option could be to dedicate each node to the ICs of a specific table (and its depending tables) and so, in the end, we just need to merge the results and as they are related to different tables that are not related between them, no inconsistency should arise. However, this suggestion is not very realistic yet because the solver does not distribute the work by more than one node and the results could originate a conflict, this is just an hypothesis.

Lastly, we hope to give a contribution to the current state of the art by enhancing the already existing analysis to the interpretation of null values by considering the contributions already available, as the different semantics, for example. We also tried to overcome the lack of applications that allow the user to repair databases considering null values and also provide some extra features to help the user in the repairing process.

However, there is still some work to do as mentioned in future work so hopefully this motivates to continue to research and solve the yet existing problems also beyond this project.

Bibliography

[ABC99] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. Consistent query answers in inconsistent databases. In *Proceedings of the eighteenth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, PODS '99, pages 68–79, New York, NY, USA, 1999. ACM.

[ABC03] Marcelo Arenas, Leopoldo Bertossi, and Jan Chomicki. Answer sets for consistent query answering in inconsistent databases. *Theory Pract. Log. Program.*, 3(4):393–424, July 2003.

[Alv11] Ricardo Alves. Database Repairs with Answer Set Programming. (Master Thesis). Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, Monte da Caparica, 2011.

[AM86] Paolo Atzeni and Nicola M. Morfuni. Functional dependencies and constraints on null values in database relations. *Information and Control*, 70(1):1–31, July 1986.

[Bar01] Chitta Baral. *Knowledge Representation, Reasoning and Declarative Problem Solving with Answer sets*. Cambridge University Press, August 2001.

[BB02] Pablo Barceló and Leopoldo Bertossi. Repairing databases with annotated predicate logic. In *NMR*, pages 160–170, 2002.

[BB04] Loreto Bravo and Leopoldo Bertossi. Consistent query answering under inclusion dependencies. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, CASCON '04, pages 202–216. Carleton University, IBM Press, 2004.

[BB06] Loreto Bravo and Leopoldo Bertossi. Semantically correct query answers in the presence of null values. In *Proceedings of the 2006 international conference on Current Trends in Database Technology*, EDBT'06, pages 336–357, Berlin, Heidelberg, 2006. Carleton University, Springer-Verlag.

[BBB03] Pablo Barceló, Leopoldo Bertossi, and Loreto Bravo. Characterizing and Computing Semantically Correct Answers from Databases with Annotated Logic and Answer Sets. In *Proceedings of the 2nd international conference on Semantics in databases*, volume 2582, pages 7–33, Berlin, Heidelberg, 2003. Springer-Verlag.

[Ber11] Leopoldo Bertossi. *Database Repairs and Consistent Query Answering*. Synthesis Lectures on Data Management. Morgan & Claypool, 2011.

[BIG10] George Beskales, Ihab F. Ilyas, and Lukasz Golab. Sampling the repairs of functional dependency violations under hard constraints. *Proc. VLDB Endow.*, 3(1-2):197–207, September 2010.

[BL04] Ronald. Brachman and Hector Levesque. *Knowledge Representation and Reasoning*. The Morgan Kaufmann Series in Artificial Intelligence. Elsevier Science, 1 edition, 2004.

[CAB00] Jan Chomicki, Marcelo Arenas, and Leopoldo Bertossi. Specifying and Querying Database Repairs using Logic Programs with Exceptions. In Kacprzyk Januss Zadrozny Slawomir Andreasen Troels Larsen, Henrik L. and Henning Christiansen, editors, *In Flexible Query Answering Systems. Recent Developments*, Advances in Intelligent and Soft Computing (Book 7), pages 27–41. Springer, 2000.

[CB07] Monica Caniupan and Leopoldo Bertossi. The consistency extractor system: Querying inconsistent databases using answer set programs. In Henri Prade and V.S. Subrahmanian, editors, *Scalable Uncertainty Management*, volume 4772 of *Lecture Notes in Computer Science*, pages 74–88, Berlin Heidelberg, 2007. Springer.

[CMR09] C. Coronel, S. Morris, and Peter Rob. *Database Systems: Design, Implementation, and Management (with Bind-in Printed Access Card)*. Bpa Series. Course Technology Ptr, 9 edition, 2009.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[Eri09] John Erickson. *Database Technologies: Concepts, Methodologies, Tools and Applications*. Number vol. 1-4 in Premier reference source. Igi Global, 1 edition, 2009.

[Fit96] Melvin Fitting. *First-Order Logic and Automated Theorem Proving*. Graduate Texts in Computer Science Series. Springer-Verlag, New York, USA, 2 edition, 1996.

[FPL+01] Enrico Franconi, Antonio Laureti Palma, Nicola Leone, Simona Perri, and Francesco Scarcello. Census data repair: a challenging application of disjunctive logic programming. In *Proceedings of the Artificial Intelligence on Logic for Programming*, LPAR '01, pages 561–578, London, UK, 2001. Springer-Verlag.

[Gel07] Michael Gelfond. *In Handbook of Knowledge Representation*, chapter Answer Sets. Elsevier Science, 2007.

[GGGM97] P. Godfrey, J. Grant, J. Gryz, and J. Minker. Integrity constraints: Semantics and applications. In *Logics for Databases and Information Systems, chapter 9*, number April, pages 265–306. Kluwer, 1997.

[GKS09] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. The conflict-driven answer set solver clasp: Progress report. In *Proceedings of the 10th International Conference on Logic Programming and Nonmonotonic Reasoning*, LPNMR '09, pages 509–514, Berlin, Heidelberg, 2009. Springer-Verlag.

[GL88] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. pages 1070–1080. MIT Press, 1988.

[GL91] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.

[GS] Kaminski Roland Kaufmann Benjamin Ostrowski Max Schaub Torsten Gebser, Martin and Thiele Sven. *A User's Guide to gringo, clasp, clingo, and iclingo*. University of Potsdam.

[GT09] Kaminski Roland Ostrowski Max Schaub Torsten Gebser, Martin and Sven Thiele. On the input language of asp grounder gringo. In *LPNMR*, pages 502–508, 2009.

[GZ03] Greco Sergio Greco, Gianluigi and Ester Zumpano. A logical framework for querying and repairing inconsistent databases. *IEEE Transactions on Knowledge and Data Engineering*, 15(6):1389–1408, November 2003.

[JSVdC12] J. Janssen, S. Schockaert, D. Vermeir, and M. de Cock. *Answer Set Programming for Continuous Domains: A Fuzzy Logic Approach*, volume 5 of *Atlantis Computational Intelligence Systems*. Atlantis Press, 2012 edition, 2012.

[KL09] Solmaz Kolahi and Laks V. S. Lakshmanan. On approximating optimum repairs for functional dependency violations. In *Proceedings of the 12th International Conference on Database Theory*, volume 361 of *ICDT '09*, pages 53–62, New York, NY, USA, 2009. University of British Columbia, ACM.

[LB07] Andrei Lopatenko and Loreto Bravo. Efficient Approximation Algorithms for Repairing Inconsistent Databases. pages 216–225. Free Univ. of Bozen-Bolzano, Bolzano, Ieee, 2007.

[Lif08] Vladimir Lifschitz. What is answer set programming? In *AAAI*, pages 1594–1597, 2008.

[LM98] A.C. Lorents and J.N. Morgan. *Database Systems: Concepts, Management and Application*. Dryden Press Series in Information Systems Series. Dryden Press, 1998.

[Mel03] Jim Melton. Information technology — Database languages — SQL — Part 2: Foundation (SQL/Foundation). *International Organization for Standardization*, 10, 2003.

[MT99] Victor W. Marek and Miroslaw Truszczynski. Stable models and an alternative logic programming paradigm. In *In The Logic Programming Paradigm: a 25-Year Perspective*, pages 375–398. Department of Computer Science, University of Kentucky, USA, Springer-Verlag, 1999.

[MT10] Ricca Francesco Manna, Marco and Giorgio Terracina. Optimized encodings for consistent query answering via asp from different perspectives. In *CILC*, 2010.

[RG02] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill international editions: Computer science series. McGraw-Hill Companies, Incorporated, 3 edition, 2002.

[SKS10] A. Silberschatz, H. Korth, and S. Sudarshan. *Database System Concepts*. Connect, learn, succeed. McGraw-Hill Companies, Incorporated, 6 edition, 2010.

[SMG10] Emanuel Santos, João Pavão Martins, and Helena Galhardas. An argumentation-based approach to database repair. In *Proceedings of the 2010 conference on ECAI 2010: 19th European Conference on Artificial Intelligence*, pages 125–130, Amsterdam, The Netherlands, The Netherlands, 2010. IOS Press.